Near Data Processing for Efficient and Trusted Systems

by

Shaizeen Aga

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Associate Professor Satish Narayanasamy, Chair
Professor Todd Austin
Assistant Professor Reetuparna Das
Professor Mingyan Liu
Professor Scott Mahlke

Shaizeen Aga
shaizeen@umich.edu
ORCID iD: 0000-0001-9552-0508

To my parents for their courage to defy norms:

My mother, Sherbanu Aga

My father, Dilawarhusen Aga

ACKNOWLEDGEMENTS

The path from dreams to success does exist.

May you have the vision to find it,

the courage to get on to it,

and the perseverance to follow it.

Wishing you a great journey.

-Kalpana Chawla, Indian-American Astronaut

Her message aboard the space shuttle Columbia to students of her college in India.

First and foremost, I would like to express my gratitude to my parents. They have been a constant source of support and encouragement throughout my life. It is thanks to their courage to defy social norms which discouraged letting a girl child pursue education that I am here today. Many thanks also to my school and undergraduate professors who helped me believe in myself.

This dissertation is a product of countless interactions with my advisor Professor Satish Narayanasamy. I am grateful to him for encouraging me to pursue a doctorate when I wasn't very confident in my ability to conduct research. He has helped and guided me over the entire course of my graduate studies. He has contributed immensely in my transformation into a computer scientist who can think critically, express ideas crisply and for that I

will forever be grateful to him.

I am also grateful to my amazing committee members Professor Todd Austin, Professor Reetuparna Das, Professor Mingyan Liu and Professor Scott Mahlke for their insightful comments on my research and for taking time out to evaluate my dissertation. Specifically, Professor Reetuparna Das advised me on my Compute Caches project and her enthusiasm for this work helped me overcome various hurdles.

I have had the pleasure to collaborate with many worthy researchers and fellow graduate students over the course of my graduate studies. I am grateful to Sriram Krishnamoorthy from Pacific Northwest National Labs for his patience and help during my CilkSpec project. I am also grateful to Abhay Singh for countless discussions on memory models and his ability to discuss any technical topic under the sun with ease. Thanks to Supreet Jeloka who helped me understand the intricacies of SRAMs and Arun Subramaniyan for his help with Compute Caches project. Many thanks to Byoungchan Oh for helpful discussions on memories and Salessawi Ferede Yitbarek for always being encouraging and being ever so ready to discuss new research ideas. Finally, thanks to my lab-mates Chun-Hung Hsiao and Subarno Banerjee for many useful discussions.

Over the course of my time at University of Michigan, I have been blessed to have an amazing group of friends who have helped me immensely. Thanks to Daya for being super-helpful and for always being available to answer my questions. Thanks to Ankit for being a constant source of support and his poor jokes. Thanks to Divya and Megha for amazing food.

Last but not the least I would like to sincerely thank my extended family who make life worth living. Neel, for showing me the bright side of things and endeavoring to make me

happy. My amazing roommate Komal for countless discussions on surviving graduate life. My super supportive set of friends Trupti and Akshay for whole heartedly believing in my ability to achieve anything I put my mind to.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Near Data Processing for Efficient and Trusted Systems

by

Shaizeen Aga

Chair: Satish Narayanasamy

We live in a world which constantly produces data at a rate which only increases with time. Conventional processor architectures fail to process this abundant data in an efficient manner as they expend significant energy in instruction processing and moving data over deep memory hierarchies. Furthermore, to process large amounts of data in a cost effective manner, there is increased demand for remote computation. While cloud service providers have come up with innovative solutions to cater to this increased demand, the security concerns users feel for their data remains a strong impediment to their wide scale adoption.

An exciting technique in our repertoire to deal with these challenges is near-data processing. Near-data processing (NDP) is a data-centric paradigm which moves computation to where data resides. This dissertation exploits NDP to both process the data deluge we face efficiently and design low-overhead secure hardware designs.

To this end, we first propose Compute Caches, a novel NDP technique. Simple augmen-

tations to underlying SRAM design enable caches to perform commonly used operations. In-place computation in caches not only avoids excessive data movement over memory hierarchy, but also significantly reduces instruction processing energy as independent sub-units inside caches perform computation in parallel. Compute Caches significantly improve the performance and reduce energy expended for a suite of data intensive applications.

Second, this dissertation identifies security advantages of NDP. While memory bus side channel has received much attention, a low-overhead hardware design which defends against it remains elusive. We observe that smart memory, memory with compute capability, can dramatically simplify this problem. To exploit this observation, we propose InvisiMem which uses the logic layer in the smart memory to implement cryptographic primitives, which aid in addressing memory bus side channel efficiently. Our solutions obviate the need for expensive constructs like Oblivious RAM (ORAM) and Merkle trees, and have one to two orders of magnitude lower overheads for performance, space, energy, and memory bandwidth, compared to prior solutions.

This dissertation also addresses a related vulnerability of page fault side channel in which the Operating System (OS) induces page faults to learn application's address trace and deduces application secrets from it. To tackle it, we propose Sanctuary which obfuscates page fault channel while allowing the OS to manage memory as a resource. To do so, we design a novel construct, Oblivious Page Management (OPAM) which is derived from ORAM but is customized for page management context. We employ near-memory page moves to reduce OPAM overhead and also propose a novel memory partition to reduce OPAM transactions required. For a suite of cloud applications which process sensitive data we show that page fault channel can be tackled at reasonable overheads.

CHAPTER I

Introduction

We are constantly being bombarded with new data everyday from varied sources. Nearly 90% of data that exists today has been produced in just the last two years, and we continue to generate nearly 2.5 quintillion bytes per day [3]. Analyzing this data holds the key to answering many unsolved problems, from accurate weather prediction to personalized medicine.

This data deluge, however, has exposed us to several challenges that need addressing. First, it has brought to light severe inefficiencies in conventional architectures which were not designed with this unprecedented data deluge in mind. Second, it has brought forth new pressing needs like increased demand for secure remote computations that need efficient solutions. The focus of this dissertation is to identify these inefficiencies, new pressing needs and address them efficiently with innovative architectural solutions.

An important shortcoming of conventional architectures that we focus on in this work is their compute centric nature. Today's conventional architectures spend large amounts of energy and time in instruction processing and data movement over deep memory hierarchies and very little on actual computation. Vector units available today only reduce

instruction processing overhead to some extent. This makes conventional architectures ill-equipped to run current data centric applications which process large amounts of data. Given the wealth of interesting insights this data holds, it is paramount to design architectures which address these inefficiencies and which can process humongous amounts of data in an efficient manner both from performance and energy perspective.

Furthermore, there is an increased demand for remote computation to process the data explosion we are facing in a cost effective manner. Cloud computing solutions are being increasingly looked up to in order to meet this demand. While cloud service providers are innovating at a lightning pace in the breadth of services they provide, from large graph processing to genetic data processing, the security concerns users feels for their data remains a strong impediment to adoption of cloud computing. As a consequence, there is a pressing need for low-overhead hardware designs which provide strong privacy and security guarantees.

An exciting technique in our repertoire to deal with these challenges is near-data processing (NDP). Prior works have observed that moving computation to where data is (as is done in NDP) is one way we can tackle the data deluge we are facing today. The research in this dissertation, however, harnesses NDP in novel ways and brings out hitherto untapped benefits of this paradigm.

To this end, we first push NDP to a new extreme where we not only move computation to where data is but transform existing memory elements (caches) into computation units. Second, this dissertation also shows that NDP not only has energy and performance efficiency benefits but it can also help us provide security guarantees in an efficient manner. As such, the research in this dissertation proposes low-overhead defenses to tackle memory

bus side channel and page fault side channel. Next, we give a brief overview of each of these solutions.

1.1 Compute Caches: Efficient Very Large Vector Processing

This dissertation identifies a novel way to exploit the NDP paradigm by enabling processor caches to perform computation. We propose the Compute Cache architecture which enables in-place computation in caches. To do so, Compute Caches use the emerging bit-line SRAM circuit technology [73, 76] to re-purpose existing cache elements and transform them into active very large vector computational units. Using such a transformation helps us support several operations: copy, search, compare and logical operations (and, or, xor, and not) which are widely used primitives and can help accelerate a wide variety of applications.

Efficiency of Compute Caches arises from two main sources: massive parallelism and reduced energy due to data movement. A cache is typically organized as a set of smaller memory arrays. A typical high-end processor has thousands of these memory arrays. All these memory arrays can potentially compute concurrently on data items stored in them, enabling us to efficiently exploit large scale data-level parallelism (e.g., 30 MB of cache can be repurposed as 960K bit-serial computation units). Furthermore, since the computation is done in-place within a cache sub-array, without transferring data in or out of it, the energy and performance overhead incurred in on-chip data movement — transferring data to and from the core, interconnects, and various levels of memory hierarchy — is drastically reduced.

We address several challenges in realizing the Compute Cache architecture. We discuss instruction set extensions and system software support needed to realize Compute Caches. We identify and solve a new problem unique to Compute Caches that we refer to as *operand locality*, which is borne out of the requirement that the data operands be stored in caches such that they share the same set of bit-lines. We also discuss simple solutions to problems such as managing parallelism across various cache levels and banks that arise as a consequence of integrating compute capable caches into a conventional cache hierarchy while preserving properties such as coherence, consistency and reliability. To support Compute Cache operations without operand locality, we also study near-place processing in cache.

We re-designed several important applications (text processing, databases, checkpointing) to utilize Compute Cache operations. We demonstrate significant speedup (1.9×) and energy savings (2.4×) compared to processors with conventional SIMD units. While our savings for applications are limited by the fraction of their computation that can be accelerated using Compute Caches (Amdahl's law), our micro-benchmarks demonstrate that applications with larger fraction of Compute Cache operations could benefit even more (54× throughput, 9× dynamic energy savings).

1.2 InvisiMem: Smart Memory Defenses for Memory Bus Side Channel

While performance and energy benefits of NDP have been exploited before in alternate ways, this dissertation also identifies security advantages that NDP can furnish. Privacy concerns is one of the strongest impediment to wider adoption of cloud computing. An

important security concern for trusted cloud computing is the memory bus side channel. An attacker can learn sensitive information about an application by observing data and addresses over the memory bus. While it has received much attention, a low-overhead solution for it is still beyond reach.

Traditional solutions to fix this vulnerability employ the Oblivious RAM (ORAM) [60] construct to hide leaking addresses over the bus. ORAM is a cryptographic construction that obfuscates memory accesses and makes them indistinguishable from a random access pattern. A secure processor can implement ORAM by issuing several memory accesses for every ORAM access. Depending on the size of ORAM, an ORAM access may incur two to three orders of magnitude increase in memory bandwidth and latency compared to a normal DRAM access. In spite of recent advancements [111, 56, 93, 160], even with significant custom hardware support [56], an ORAM-enabled secure processor increases memory access latency by 20X, which can result in a performance overhead of about 4X. Beside hiding addresses accessed, it is also paramount to hide memory access times, memory address trace length and provide other security guarantees like data freshness. Providing these additional guarantees only adds to existing severe ORAM overheads.

This dissertation observes that smart memory, memory with compute capability and a packetized interface, can dramatically simplify this problem. Such smart memories are possible today due to recent advancements in 3D integration technology such as the Hybrid Memory Cube (HMC) [12] which make it possible to stack DRAM layers on top of logic layers, and connect them using Through Silicon Via (TSV).

Our proposed solution InvisiMem expands the trust base to include the logic layer in the smart memory to implement cryptographic primitives, which aid in addressing several

memory bus side channel vulnerabilities efficiently. This allows the secure host processor to send encrypted addresses over the untrusted memory bus, and thereby eliminates the need for expensive address obfuscation techniques based on ORAM. Additional measures are required for address confidentiality which we identify and implement efficiently. In addition to address confidentiality, we also discuss how smart memory helps mitigate memory bus timing channel using constant heart-beat packets.

Furthermore, we also observe that smart memory can help reduce the overhead incurred for guaranteeing data freshness. An adversary tapping the memory bus can rollback the state of a memory block by using older messages. To defeat such replay attacks, the secure processor needs to maintain additional state (version numbers) to ensure that a read response returns the latest version for a memory block ([152, 56]). With compute capability in memory, InvisiMem establishes a secure channel of communication between the secure processor and memory, such that it guarantees freshness without maintaining such version numbers in the secure host processor.

We demonstrate that InvisiMem designs provide strong defenses against memory bus side channels and efficient data freshness guarantee and have one to two orders of magnitude of lower overheads for performance, space, energy, and memory bandwidth, compared to prior solutions.

1.3 Sanctuary: Secure and Efficient Memory Management

This dissertation also harnesses NDP to lower the overheads for page fault channel defense. Current state of the art secure processors like Intel SGX (Software Guard Exten-

sions) leave virtual memory management to the operating system. An untrusted operating system can exploit this to induce page faults on every memory access by the application. By doing this, a malicious OS can learn memory access trace of the application at page granularity. A recent work [151] has demonstrated how an application's page access pattern obtained via page-fault side channel can be used to deduce the program path it took during execution which in turn can help deduce the application's sensitive inputs or outputs. Specifically, they showed how the input image to an image processing application can be completely recovered. This can have catastrophic consequences for privacy of confidential data like medical images which are being processed by cloud services.

Current solutions to fix page fault channel [40] make the unrealistic assumption that all memory needed by an application is reserved a priori. Predetermining application memory requirement is possible by either severely limiting application behavior (no dynamic memory allocations, no recursion etc.) or by reserving large amounts of memory a priori. The latter case can cause information leak if application memory requirement exceeds the reserved memory size during runtime, leading to OS controlled paging activity. Finally, yet importantly, these solutions rob the OS of it's flexibility in managing memory as a resource; once allocated, memory cannot be reclaimed.

To address the insufficiency of prior works, this dissertation presents Sanctuary, a novel page fault channel defense which unlike prior solutions preserves operating system's flexibility in managing memory as a resource by allowing on-demand page allocations and deallocations. We design a secure runtime which interfaces with the OS on behalf of the application to hide addresses accessed by the application and also secures address translations for sensitive application pages. Our secure runtime uses a novel construct, Oblivious

Page Management (OPAM) which is derived from Oblivious RAM construct [60] but is customized for addressing challenges and exploiting opportunities that arise in the context of page management. The chief source of OPAM overhead comprises of page moves and we perform these page movements near-memory to lower their overheads. Finally, we also propose a novel memory partition which helps reduce the OPAM transactions needed in our system and further brings down the overheads of our solution. We study a suite of cloud applications and show that page fault channel can be fixed at reasonable overheads.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter II presents some background material on near data processing. Chapter III presents Compute Caches which enables computation in caches and delivers performance and energy wins. Chapter IV presents InvisiMem, a secure processor which solves memory bus side channels efficiently using 3D stacked memories. Chapter V discusses an efficient defense against page fault channel vulnerability. Finally, in Chapter VI we conclude the dissertation by summarizing our contributions.

CHAPTER II

Background

In this chapter we look at prior work on Near Data Processing (NDP) paradigm to put the research presented in this proposal in proper context. NDP is a data centric computation paradigm in which computation is moved to where data resides. Prior work on Near Data Processing can be divided into two parts based on whether or not the proposals employ 3D Stacked DRAM. We first present prior work which does not employ such "smart" memories. We then give a brief background on 3D stacked smart memory followed by proposals which do employ them to harness performance and energy efficiency. Finally, we talk about how the research presented in this proposal breaks new ground in both expanding the reach of Near Data Processing and identifying its new advantages which have been hitherto unexplored.

2.1 NDP proposals without 3D Stacked Memory

Traditional DRAM chips have abundant internal bandwidth thanks to large row buffers but only a tiny amount of data per access is shipped to processor (cache block). To solve

this, EXECUBE [86] augments a standard, low cost DRAM chip with logic array comprising of eight processors and interface ports. Each such processor can either work independently or in tandem with other processors in a SIMD fashion. Such processors then enjoy far more memory bandwidth than traditional systems which connect to off-chip memory via memory bus.

Terasys [58] transforms conventional memory into a SIMD processor array. In this proposal, a traditional SRAM array can either support normal read/write mode or a special PIM (processing in memory) mode. In PIM mode, processors associated with each column in SRAM array perform a specific command on a selected row in SRAM array. The processors present are bit serial that access and process bits from attached memory. Each cycle, the processors can either load or store data and perform computation on data using an ALU. Terasys also provides some reduction capabilities on the results produced by these processors such as combining results of all processors to send a single bit result to host processor.

IRAM [106] proposes unifying DRAM and logic into single DRAM chip in response to growing processor-memory performance gap. Such an organization can utilize internal DRAM bandwidth better as compared to traditional processor memory configuration. As memory is now effectively on-chip, memory latency is lower. Furthermore, energy expended is lower too as far more memory is available on-chip than would be possible using SRAM as DRAM is far denser.

Active Pages [102] proposes partitioning applications between traditional processor and intelligent memory system such that data intensive parts of the application are carried out by the memory. “Active pages” consists of a page of data and functions which manipulate the

data. The memory system employed is capable of both storing the data and performing the data manipulations. This proposal implements memory system managing "active pages" using re-configurable architecture DRAM which integrates FPGA and DRAM technology.

FlexRAM [77] models memory flexibly as either default DRAM chip or a PIM (processing in memory) chip which performs computation. This proposal models PIM chips based on Merged Logic DRAM (MLD) process which integrates logic and DRAM in a single chip. PIM chips consist of simple processing elements finely interleaved with DRAM cells which have high bandwidth access to memory. A low-issue superscalar RISC core is also modeled in each PIM chip to co-ordinate the processing of the simple processing elements without relying on host processor.

DIVA [49] or Data IntensiVe Architecture models PIM chips (memory with processing logic) as co-processors for a conventional system which can either be used to perform computation close to memory or be used as traditional DRAM chips. Each DIVA PIM chip is a memory device augmented with a processor and communication hardware for PIM chips to interact with each other. Form of active messages called "parcels" are used to communicate computation and synchronize PIM chips. DIVA memory is partitioned between that accessible to host processor only, PIM chips only and global memory that is accessible to both. DIVA also enables light weight address translations on PIM chips using segments.

While all prior proposals aim to add logic to memory to support general purpose programs, Intelligent Memory Manager [112] augments memory with a simple processor which solely aims to aid the host processor by performing memory management functions (prefetching of data, relocation of data) on its behalf. Such functions typically tend to

pollute processor caches. By performing them using processor in memory, such pollution can be avoided.

2.2 3D Stacked Memory

Advancements in 3D integration have made possible 3D Stacked memories which are now commercially available [16]. A typical 3D Stacked memory consists of several layers of DRAM dies stacked on top of each other, with a logic layer at the bottom. The DRAM banks are organized into sets of independent vaults, and the various layers are connected using Through Silicon Vias (TSVs) [12]. High density, short length TSVs in conjunction with vaults which function independently and can be accessed in parallel are the chief sources of high bandwidth provided by such a system.

Two of the most popular 3D stacked designs present are HMC [12] and HBM [83]. Both of them employ multiple memory dies stacked together with a logic die at the base. HMC places memory controller logic in the logic die. While their internal implementations differ, most importantly, how they are integrated with the main host processor also differs. In case of HMC an high speed packet based serial interface connects it to the CPU. In case of HBM however, interaction with CPU is achieved using another silicon layer, termed silicon interposer.

2.2.1 NDP proposals with 3D Stacked Memory

Several proposals employ such 3D Stacked memories to harness performance for various applications. Pugsley*et al.* [109] model several simple processing cores in the logic

layer of 3D Stacked memory which perform the Map operation to accelerate MapReduce workloads. In TOP-PIM [159], the authors model GPU in logic layer of 3D Stacked memory and study several GPGPU workloads to identify characteristics that make workloads amenable to be offloaded to GPU near memory. Ahn *et al.* [22] proposed a locality aware PIM design, which opportunistically decides when computation should be offloaded to memory and when it should be performed using host processor instructions.

2.3 Breaking New Ground in NDP

All prior proposals augment the main memory to play an active part in computation. For applications with no cache locality, compute capable memory is clearly an effective solution. But for cache friendly applications, always computing in memory may not be an efficient choice. However, the other extreme of computing the traditional way, as this proposal shows, incurs high instruction processing overheads and expends significant energy for moving data over memory hierarchy.

To address these limitations, this proposal introduces a new way to realize near-data computing by enabling processor caches to perform computation. Using our proposed solution, caches can perform simple operations common to wide variety of applications. Using our system, data can be moved to the cache level most suitable to it's reuse profile.

Most near-data processing proposals presented in this chapter seek to attain performance and energy gains. Our proposal also identifies and exploits an hitherto untapped advantage of NDP to provide low-overhead security guarantees.

CHAPTER III

Compute Caches: Caches that Compute

This chapter presents Compute Caches, our novel proposal to enable computations in processor caches. With Compute Caches we add a new technique to exploit the near data processing paradigm. Compute centric architectures employ processor caches merely to stage data and they otherwise play no active role in computation. However, with Compute Caches, processor caches can now play an active and a powerful role in computation. Compute Caches not only provide advantages of vector processing but also help avoid excessive data movement over memory hierarchy.

3.1 Introduction

As computing today is dominated by data-centric applications, there is a strong impetus for specialization for this important domain. Conventional processors' narrow vector units fail to exploit the high degree of *data-parallelism* in these applications. Also, they expend disproportionately large fraction of time and energy in *moving data* over cache hierarchy, and in instruction processing, as compared to the actual computation [41].

We present the Compute Cache architecture for dramatically reducing these inefficiencies through in-place (in-situ) processing in caches. A modern processor devotes a large fraction (40-60%) of die area to caches which are used for storing and retrieving data. Our key idea is to re-purpose and transform the elements used in caches into active computational units. This enables computation in-place within a cache sub-array, without transferring data in or out of it. Such a transformation can unlock massive data-parallel compute capabilities, dramatically reduce energy spent in data movement over the cache hierarchy, and thereby directly address the needs of data-centric applications.

Our proposed architecture uses an emerging SRAM circuit technology, which we refer to as *bit-line computing* [73, 76]. By simultaneously activating multiple word-lines, and sensing the resulting voltage over the shared bit-lines, several important operations over the data stored in the activated bit-cells can be accomplished without data corruption. A recently fabricated chip [73] demonstrates feasibility of bit-line computing. They also show a stability of more than six sigma robustness for Monte Carlo simulations, which is considered industry standard for robustness against process variations.

Past processing-in-memory (PIM) solutions proposed to move processing logic *near* the cache [87, 50] or main memory [107, 126]. 3D stacking can make this possible [22]. Compute Caches significantly push the envelope by enabling *in-place* processing using existing cache elements. It is an effective optimization for data-centric applications, where at least one of the operands (e.g., dictionary in WordCount) used in computation has cache locality.

Efficiency of Compute Caches arises from two main sources: massive parallelism and reduced data movement. A cache is typically organized as a set of sub-arrays; as many

as hundreds of sub-arrays, depending on the cache level. These sub-arrays can potentially compute concurrently on data stored in them (KBs of data) with little extensions to the existing cache structures (8% of cache area overhead). Thus, caches can effectively function as large vector computational units, whose operand sizes are orders of magnitude larger than conventional SIMD units (KBs vs bytes). To achieve similar capability, the logic close to memory in a conventional PIM solution would need to provision more than hundred additional vector functional units. The second benefit of Compute Caches is that they avoid the energy and performance cost incurred not only for transferring data between the cores and different levels of cache hierarchy (through network-on-chip), but even between a cache's sub-array to its controller (through in-cache interconnect).

We address several problems in realizing the Compute Cache architecture, discuss ISA and system software extensions, and re-designs several data-centric applications to take advantage of the new processing capability.

An important problem in using Compute Caches is satisfying the *operand locality* constraint. Bit-line computing requires that the data operands are stored in rows that share the same set of bit-lines. We architect a cache geometry, where ways in a set are judiciously mapped to a sub-array, so that software can easily satisfy operand locality. Our design allows a compiler to ensure operand locality simply by placing operands at addresses that are page aligned (same page offset). It avoids exposing the internals of a cache, such as its size or geometry, to software.

When in-place processing is not possible for an operation due to lack of operand locality, we propose to use *near-place* Compute Caches. In near-place design, the source operands are read out from the cache sub-arrays, the operation is performed in a logic unit

placed close to the cache controller, and the result may be written back to the cache.

Besides operand locality, Compute Caches brings forth several interesting questions. How to orchestrate concurrent computation over operands spreading across multiple cache sub-arrays? How to ensure coherence between compute-enabled caches? How to ensure consistency model constraints when computation is spread between cores and caches? Soft errors are a significant concern in modern processors. Can Error Correction Codes (ECC) be used for Compute Caches? When not possible, what are the alternative solutions? We discuss relatively simple solutions to address these problems.

Compute Caches support several in-place vector operations: copy, search, compare and logical operations (and, or, xor, and not) which can accelerate a wide variety of applications. We study two text processing applications (word count, string matching), database query processing with bitmap indexing, copy-on-write checkpointing in OS, and bit matrix multiplication (BMM); a critical primitive used in cryptography, bioinformatics, and image processing. We re-designed these applications to efficiently represent their computation in terms of Compute Cache supported vector operations. Section 3.5 identifies a number of additional domains that can benefit from Compute Caches: data analytics, search, network processing etc.

We evaluate the merits of Compute Caches for a multi-core processor modeled after Intel's SandyBridge [88] processor with eight cores, three levels of caches, and a ring interconnect. For the applications we study, on average, Compute Caches improve performance by 1.9× and reduce energy by 2.4× compared to a conventional processor with 32-byte wide vector units. Applications with a higher fraction of Compute Cache operations can benefit significantly more. Through micro-benchmarks that manipulate 4KB operands, we

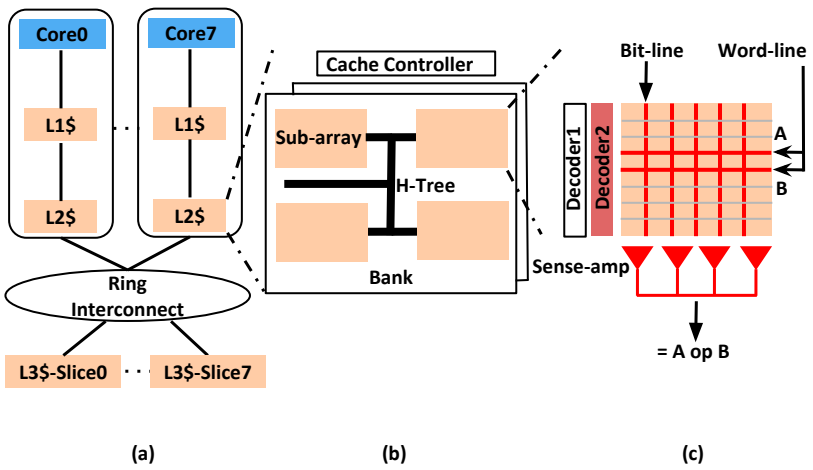


Figure 3.1: Compute Cache overview. (a) Cache hierarchy. (b) Cache geometry (c) In-place compute in a sub-array.

show that Compute Caches provide 9× dynamic energy savings over a baseline using 32-byte SIMD units while providing 54× better throughput on average.

3.2 Background

This section provides a brief background of cache hierarchy, cache geometry, and bit-line computing in SRAM.

3.2.1 Cache Hierarchy and Geometry

Figure 3.1 (a) illustrates a multi-core processor modeled loosely after Intel's Sandy-bridge [88]. It has a three-level cache hierarchy comprising of private L1 and L2, and a shared L3. The shared L3 cache is distributed into slices which are connected to the cores via a shared ring interconnect. A cache consists of a cache controller and several banks ((Figure 3.1 (b)). Each bank has several sub-arrays connected by a H-Tree interconnect. For example, a 2 MB L3 cache slice has a total of 64 sub-arrays distributed across 16 banks.

A sub-array in a cache bank is organized into multiple rows of data-storing bit-cells.

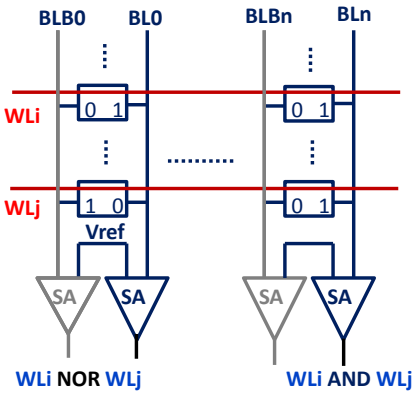


Figure 3.2: SRAM circuit for in-place operations. Two rows (WL_i and WL_j) are activated. An *AND* operation is performed by sensing bit-line (BL). All the bit-lines are initially pre-charged to '1'. If both the activated bits in a column have a '1' (column 'n'), then the BL stays high and it is sensed as a '1'. If any one of the bits were '0' it will lower the BL voltage below V_{ref} and will be sensed as a '0'. A *NOR* operation can be performed by sensing bit-line bar (BLB).

The bit-cells in the same row are connected to a word-line. The bit-cells along a column share the same bit-line. Typically, in any cycle, one word-line is activated, from where a data block is either read from, or written to, through the column bit-lines.

3.2.2 Bit-line Computing

Compute Caches use emerging bit-line computing technology in SRAMs [73, 76] (Figure 3.2) which observes that, when multiple word-lines are activated simultaneously, the shared bit-lines can be sensed to produce the result of `and` and `nor` on the data stored in the two activated rows. Data corruption due to multi-row access is prevented by lowering word-line voltage to bias against write of the SRAM array. Jeloka *et al.* [73]'s measurements across 20 fabricated test chips demonstrate that data-corruption does not occur even when 64 word-lines are simultaneously activated during such an in-place computation. They show a stability of more than six sigma robustness for Monte Carlo simulations, which is considered industry standard for robustness against process variations. Also, note

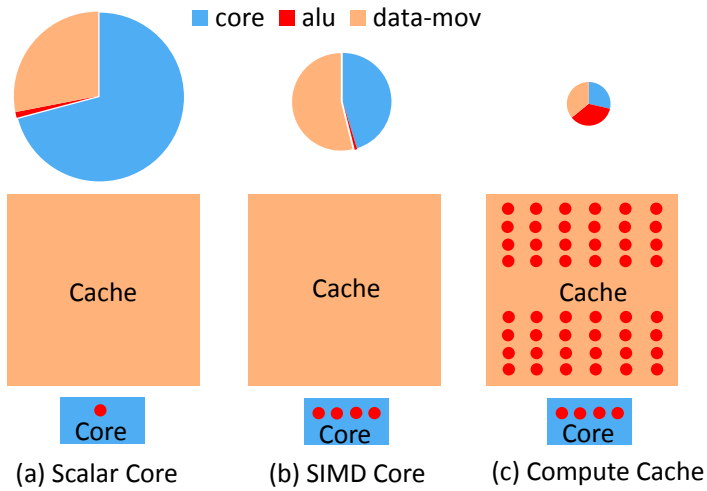


Figure 3.3: Proportion of energy (top) for bulk comparison operation and area (bottom). Red dot depicts logic capability.

that, by lowering the word-line voltage further, robustness can be improved at the cost of increase in delay. Even with it, Compute Caches will still deliver significant savings given its potential (Section 3.6, 54× throughput, 9× dynamic energy savings).

Section 3.4.2 discusses our extensions to bit-line computing enabled SRAM to support additional operations: `copy`, `xor`, equality comparison, search, and carryless multiplication (`clmul`).

3.3 A Case for Compute Caches

In-place Compute Cache has the potential to provide massive data-parallelism, while also dramatically reducing the instruction processing and on-chip data movement overheads. Figure 3.3 pictorially depicts these benefits by comparing a scalar core, a SIMD core with vector processing support, and Compute Caches.

The bottom half in Figure 3.3 depicts the area proportioning and processing capability of the three architectures. Significant fraction of die area in a conventional processor is for

Cache	cache-ic (h-tree)	cache-access
L1-D	179 pJ	116 pJ
L2	675 pJ	127 pJ
L3-slice	1985 pJ	467 pJ

Table 3.1: Cache energy per read access

caches. A Compute Cache re-purposes the elements used in this large area into compute units for a small area overhead (8% of cache area). A typical last-level cache consists of hundreds of sub-arrays distributed across different banks which can potentially compute concurrently on cache blocks stored in them. This enables us to exploit large scale data level parallelism (e.g. a 16MB L3 has 512 sub-arrays and can support 8 KB operands) dwarfing even a SIMD core.

The top row of Figure 3.3 shows relative energy consumption for a comparison operation over several blocks of 4KB operands (Section 3.6.4). In a scalar core, less than 1% of the energy is expended on the ALU operation, while nearly three quarters of the energy is spent in processing instructions in the core, and one-fourth is spent on data movement. While vector processing (SIMD) support (Figure 3.3 (b)) in general-purpose and data-parallel accelerators reduce the instruction processing overhead to some degree, it does not help address the data movement overhead. Compute Cache architecture (Figure 3.3 (c)) can reduce the instruction processing overheads by an order of magnitude, by supporting SIMD operation on large operands (tens of KB). Also, it avoids the energy and performance cost due to data movement.

In-place Compute Cache reduces on-chip *data movement overhead*, which consists of two components. First, is the energy spent on data transfer. This includes not only the significant energy spent on the processor interconnect's wires and routers, but also the

H-Tree interconnect used for data transfer within a cache. A near-place Compute Cache solution can solve the former but not the latter. As shown in Table 3.1, H-Tree consumes nearly 80% of cache energy spent in reading from a 2MB L3 cache slice.

Second, is the energy spent when reading and writing in the higher-level caches. In a conventional processor, a data block trickles up the cache hierarchy all the way from L3 to L1 cache, and into a core's registers, before it can be operated upon. An L3 Compute Cache can eliminate all this overhead. A shared L3 Compute Cache can also reduce the cost of sharing data between two cores, as it would avoid write-back from a source core's L1 to shared L3, and then a transfer back to a destination core's L1.

3.4 Compute Cache Architecture

Figure 3.1 illustrates the Compute Cache (CC) architecture. We enhance all the levels in the cache hierarchy with in-place compute capability. Computation is done at the highest level where the application exhibits significant locality. In-place compute is based on the bit-line computing technology we discussed in Section 3.2. We enhance these basic in-place compute capabilities to support `xor` and several in-place operations (copy, search, comparison, and carryless multiplication).

In-place computing is possible only when operands are mapped to sub-arrays such that they share the same bit-lines. We refer to this requirement as operand locality. We discuss a cache geometry that allows a compiler to satisfy operand locality by ensuring that the operands are page-aligned.

Each cache controller is extended to manage the parallel execution of CC instructions

Opcode	Src1	Src2	Dest	Size	Description
cc_copy	a	-	b	n	$b[i] = a[i]$
cc_buz	a	-	-	n	$a[i] = 0$
cc_cmp	a	b	r	n	$r[i] = (a[i] == b[i])$
cc_search	a	k	r	n	$r[i] = (a[i] == k)$
cc_and	a	b	c	n	$c[i] = a[i]$ & $b[i]$
cc_or	a	b	c	n	$c[i] = (a[i] \mid\mid b[i])$
cc_xor	a	b	c	n	$c[i] = a[i] \oplus b[i]$
cc_clmulX	a	k	c	n	$c_i = \oplus(a[i]$ & $k)$
cc_not	a	-	b	n	$b[i] =!(a[i])$
a,b,c,k: addresses			r:register	$\forall i, i \in [1, n]$, X = [64/128/256]	

Table 3.2: Compute Cache ISA.

across its several banks. It also decides the cache level to perform the computation and fetches the operands to that level. Given that a Compute Cache can modify data, we discuss its implication in ensuring coherence and consistency properties. Finally, we discuss design alternatives for supporting ECC in Compute Caches.

In the absence of operand locality, we propose to compute near-place in cache. For this, we add a Logic Unit in the cache controller. Although near-place cache computing requires additional functional units, and cannot save H-Tree interconnect energy inside caches, it successfully helps reduce the energy spent in transferring and storing data in the higher-level caches.

3.4.1 Instruction Set Architecture (ISA)

Compute Cache (CC) ISA extensions are listed in Table 3.2. It supports several vector instructions, whose operands are specified using register indirect addressing. Operand sizes are specified through immediate values and can be as large as 16K. It supports vector copying, zeroing, and logical operations. It also supports vector carry less multiply instruction (cc_clmul) at single/double/quad word granularity. It also supports equality comparison

and search. We limit the operand size (n) of these instructions to 64 words (512 bytes), so that the result can be returned as a 64-bit value to a processor core's register. For the search instruction, the key size is set to 64 bytes. For smaller keys, the programmer can either duplicate the key multiple times starting from the key's address (if its size is a word multiple), or pad the key and source data operands to be 64 bytes.

3.4.2 Cache Sub-arrays with In-Place Compute

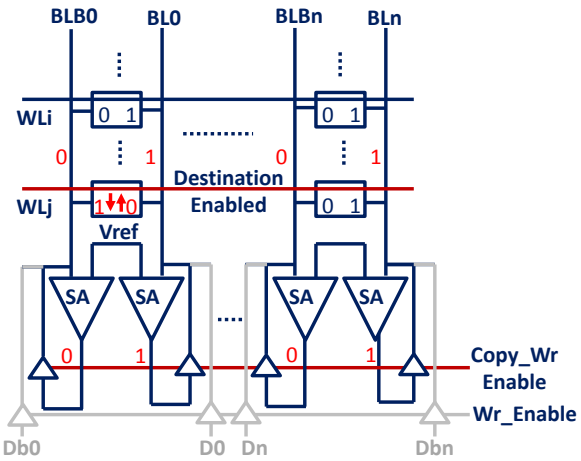


Figure 3.4: In-place copy operation (from row i to j).

Compute Caches is made possible by our SRAM sub-array design that facilitates in-place computation. We start with the basic circuit framework proposed by Jeloka *et al.* [73], which supports logical `and` and `nor` operations. To a conventional cache's sub-array, we add an additional decoder to allow activating two wordlines, one for each operand. The two single-ended sense amplifiers required for separately sensing both the bit-lines attached to a bit-cell are obtained by re-configuring the original differential sense amplifier.

In addition to `and` and `nor` operations, we extend the circuit to support `xor` operation by NOR-ing bit-line and bit-line complement. We realize compound operations such as

compare and *search* by using the results of bitwise `xor`. To compare two words, the individual bit-wise `xor` results are combined using a wired-NOR. Comparison is utilized to do iterative *search* over cache blocks stored in sub-arrays.

By feeding the result of the sense-amplifiers back to the bit-lines, one word-line can be copied to another without ever latching the source operand. We leverage the fact that the last read value is same as the data to be written in the next cycle, and coalesce the read-write operation to enable more energy-efficient *copy* operation as shown in Figure 3.4. By resetting input data latch before a write we can enable in-place zeroing of a cache block.

Finally, the carryless multiplication (`clmul`) operation is done using a logical `and` on two sub-array rows, followed by `xor` reduction of all the resultant bits. This is supported by adding a `xor` reduction tree to each sub-array.

Our extensions have negligible impact on the baseline read/write accesses as they use the same circuit as the baseline, including differential sensing. An in-place operation takes longer than a single read or write sub-array access, as it requires longer word-line pulse to activate and sense two rows to compensate for the lower word-line voltage. Sensing time also increases due to the use of single-ended sense amplifiers, as opposed to differential sensing. However, note that this is still less than the delay baseline would incur to accomplish an equivalent in-place operation, as it would require multiple read accesses and/or write access. Section 3.6.3 provides the detailed delay, energy and area parameters for compute capable cache sub-arrays.

3.4.3 Operand Locality

For in-place operations, the operands need to be physically stored in a sub-array, such that they share the same set of bitlines. We term this requirement as *operand locality*. In this section, we discuss cache organization and software constraints that can together satisfy this property. *Fortunately, we find that software can ensure operand locality as long as operands are page-aligned, i.e., have the same page offset. Besides this, the programmer or the compiler does not need to know about any other specifics of the cache geometry.*

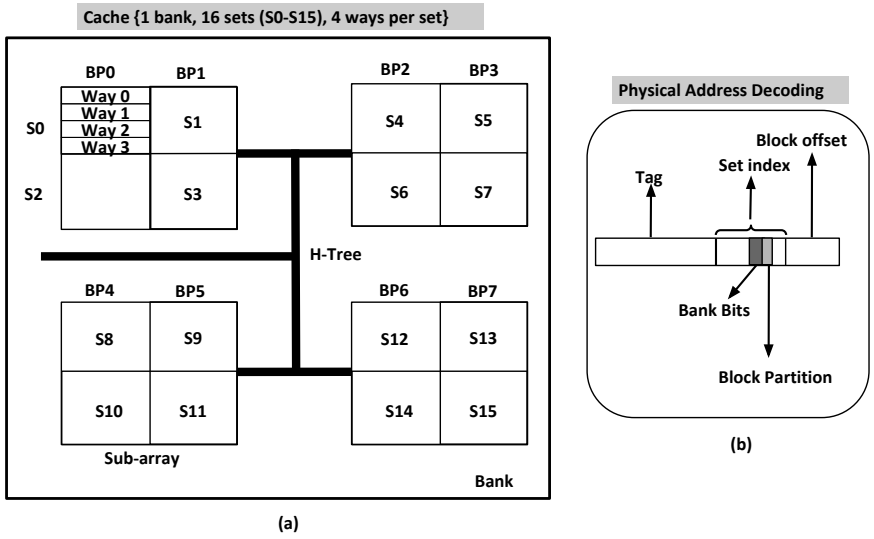


Figure 3.5: Cache organization example, address decoding ([i][j] = set i, way j), alternate address decoding for parallel tag-data access caches

Operand locality aware cache organization: Figure 3.5 illustrates a simple cache with one bank each with four sub-arrays. Rows in a sub-array share the same set of bitlines.

We define a new term, *Block Partition* (BP). Block partition for a sub-array is the group of cache blocks in that sub-array that share the same bitlines. In-place operation is possible between any two cache blocks stored within a block partition. In our example, since each row in a sub-array has two cache blocks, there are two block partitions per sub-array. In

total, there are eight block partitions (BP0-BP7). In-place compute is possible between any blocks that map to the same block partition (e.g. blocks in sets `S0` and `S2`).

We make two design choices for our cache organization to simplify operand locality constraint. First, all the ways in a set are mapped to the same block partition as shown in Figure 3.5(a). This ensures that operand locality would not be affected based on which way is chosen for a cache block.

Second, we use a portion of set-index bits to select the block's bank and block partition, as shown in Figure 3.5(b). As long as these are the same for two operands, they are guaranteed to be mapped to the same block partition.

Software requirement: The number of address bits that must match for operand locality varies based on the cache size. As shown in Table 3.3, even the largest cache (L3) in our model requires that only least 12 bits are the same for two operands (we assume pages are mapped to a NUCA slice closest to the core actively accessing them). Given that our pages are 4KB in size, we observe that as long as the operands are page aligned, i.e., have the same page offset, then they will be placed in the address space such that the least significant bits (12 for 4 KB page) in their addresses (both virtual and physical) match. This would trivially satisfy the operand locality requirement for all the cache levels and sizes we study. Note that, we only require operands to be placed at the same offset of 4KB memory regions, and it is not necessary to place them in separate pages. For super-pages that are larger than 4KB, operands can be placed within a page while ensuring 12-bit address alignment.

We expect that for data-intensive regular applications that operate on large chunks of data, it is possible to satisfy this property. Many operating system operations that involve

Cache	Banks	BP	Block size	Min. address bits match
L1-D	2	2	64	8
L2	8	2	64	10
L3-slice	16	4	64	12

Table 3.3: Cache geometry and operand locality constraint.

copying from one page to another are guaranteed to exhibit operand locality for our system. Compiler and dynamic memory allocators could be extended to optimize for this property in future.

Finally, a binary compiled with a given address bit alignment requirement (12 bits in our work) is portable across a wide range of cache architectures as long as the number of address bits to be aligned is equal to or less than what they were compiled for. If the cache geometry changes such that it requires greater alignment, then the programs would have to be recompiled to satisfy that stricter constraint.

Column Multiplexing: With column multiplexing, multiple adjacent bit-lines are multiplexed to a single bit data output, which is then observed using one sense-amplifier. This keeps area overhead of peripherals under check and improves resilience to particle strikes. Fortunately, in column multiplexed sub-arrays, adjacent bits in a cache block are interleaved across different sub-arrays such that their bitlines are not multiplexed. In this case, the logical block partition that we defined would be interleaved across the sub-arrays. Thus, an entire cache block can be accessed in parallel. Given this, in-place concurrent operation on all the bits in a cache block is possible even with column multiplexing.

Our design choice of placing ways of a set within a block partition does not affect the degree of column multiplexing as we interleave cache blocks of different sets instead.

Way Mapping vs Parallel Tag-Data Access: We chose to place all the ways of a set

within a block partition, so that operand locality is not dependent on which way is chosen for a block at runtime. However, this prevents us from supporting parallel tag-data access, where all the cache blocks in a set are pro-actively read in parallel with the tag match. This optimization is typically used for L1 as it can reduce the read latency by overlapping tag match with read. But it incurs a high read energy overhead ($4.7\times$ higher energy per access for L1 cache) for modest performance gain (2.5% for SPLASH-2[147]). Given the significant benefits of L1 Compute Cache, we think it is a worthy trade-off to forgo this optimization for L1.

3.4.4 Managing Parallelism

Cache controllers are extended to provision for CC controllers which orchestrate the execution of CC instructions. The CC controller breaks a CC instruction into multiple simple vector operations whose operands span at most a single cache block and issues them to the sub-arrays. Since a typical cache hierarchy can have hundreds of sub-arrays (16MB L3 cache has 512 sub-arrays), we can potentially issue hundreds of concurrent operations. This is only limited by two factors. First, the bandwidth of the shared interconnects used to transmit address/commands. Note that we do not replicate the address bus in our H-tree interconnects. Second, number of sub-arrays activated at same time can be limited to limit peak power drawn.

The controller at L1-cache uses an `instruction table` to keep track of the pending CC instructions. The simple vector operations are kept track of in the `operation table`. The instruction table tracks metadata associated at instruction level (i.e., result, count of simple vector operations completed, next simple vector operation to be gener-

ated). The operation table, on other hand, tracks status of each operand associated with the operation and issues request to fetch the operand if it is not present in the cache (Section 3.4.5). When all operands are in cache, we issue the operation to the cache sub-array. As operations complete they update the instruction table, and the L1-cache controller notifies the core when an instruction is complete.

To support search instruction, CC controller replicates key in all the block partitions where the source data resides. To avoid doing this again for the same instruction, we track such replications per instruction in a `key table`.

Finally, if the address range of any operand of a CC instruction spans multiple pages, it raises a pipeline exception. The exception handler splits the instruction into multiple CC operations such that each of its operands are within a page.

3.4.5 Fetching In-Place Operands

The Compute Cache (CC) controllers are responsible for deciding the level in the cache hierarchy where CC operations need to be performed, and issuing commands to the cache sub-array to execute them. To simplify our design, in our study, the CC controller always performs the operations at the highest-level cache where *all* the operands are present. If any of the operands are not cached, then the operation is performed at lowest-level cache (L3). Cache allocation policy can be improved in future by enhancing our CC controller with a cache block reuse predictor [70].

Once a cache level is picked, CC controller fetches any missing operands to that level. The controller also pins the cache-lines the operands are fetched in while the CC operation is under way. To avoid the eviction of operands while waiting for missing operands, we

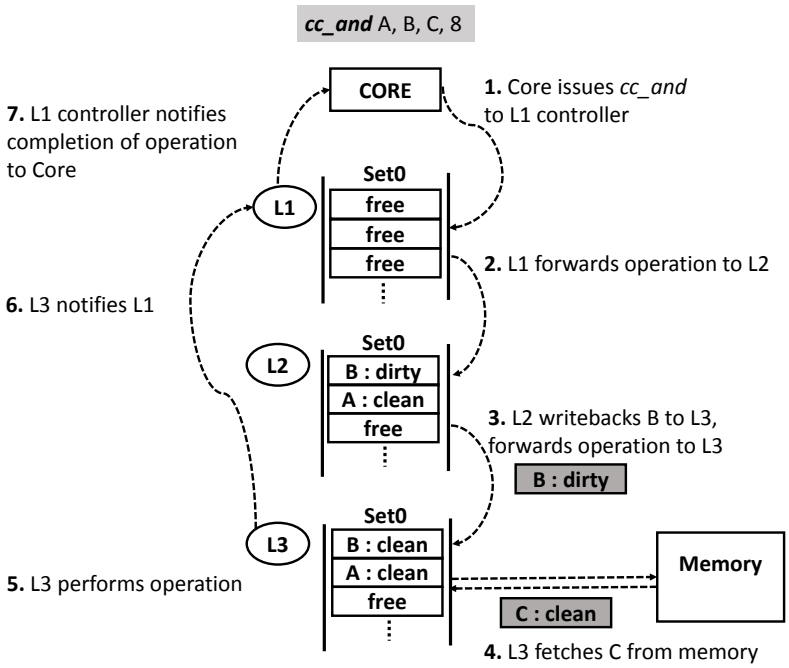


Figure 3.6: Compute Caches in action

promote the cache blocks of that operand to the MRU position in the LRU chain. However, on receiving a forwarded coherence request, we release the lock to avoid deadlock and re-fetch the operand. Getting a forwarded request to a locked cache line will be rare for two reasons. First, in DRF [20] compliant programs, only one thread will be able to operate on a cache block while holding its software lock. Second, as operands of a single CC operation are cache block wide, false sharing will be low. Nevertheless, to avoid starvation in pathological scenarios, if CC operation fails to get permission after repeated attempts (set to two), processor core will translate and execute a CC operation as RISC operations.

Figure 3.6 shows a working example. Core issues operation cc_and with address operands A, B and C to L1 controller (❶). Each is of size 64 bytes (8 words) spanning an entire cache block. For clarity, only one cache set in each cache level is shown. None of the operands are present in L1 cache. Operand B is in L2 cache and is dirty. L3 cache has

clean copy of A and a stale copy of B. C is not in any cache.

L3 cache is chosen for the CC operation, as it is the highest cache level where all operands are present. L1 and L2 controllers will forward this operation to L3 (❷, ❸). Before doing so, L2 cache will first write-back B to L3. Note that caches already write-back dirty data to next cache level on eviction and we use this existing mechanism.

On receiving the command, L3 fetches C from memory (❹). Note that, as an optimization, C need not be fetched from memory as it will be over-written entirely. Once all the operands are ready, L3 performs the CC operation (❺) and subsequently notifies the L1 controller (❻) of it's completion, which in turn notifies the core (❼).

3.4.6 Cache Coherence

Compute Cache optimization interacts with the cache coherence protocol minimally and as a result does not introduce any new race conditions. As discussed above, while the controller locks cache lines while performing CC operation, on receipt of a forwarded coherence request, the controller releases the lock and responds to the request. Thus, a forwarded coherence request is always responded to in cases where it would be responded to in the baseline design.

Typically, higher-level caches writeback dirty data to the next-level cache on evictions. Coherence protocols already support such writebacks. In the Compute Cache architecture, when a cache level is skipped to perform CC operations, any dirty operands in the skipped level need to be written back to next level of cache to ensure correctness. To do this, we use the existing writeback mechanism and thus require no change to the underlying coherence protocol.

3.4.7 Consistency Model Implications

Current language consistency models (C++ and Java) are variants of the DRF model [20], and therefore a processor only needs to adhere to the RMO memory model. While ISAs providing stronger guarantees (x86) exist, they can be exploited only by writing assembly programs. As a consequence, while we believe stronger memory model guarantees for Compute Caches is an interesting problem (to be explored in future work), we assume RMO model in our design. In RMO, no memory ordering is needed between data reads and writes, including all CC operations. Individual operations within a vector CC instruction can also be performed in parallel by the CC controller.

Programmers use `fence` instructions to order memory operations, which is sufficient in the presence of CC instructions. Processor stalls commit of a fence operation until preceding pending operations are completed, including CC operations. Similar to conventional vector instructions, it is not possible to specify a fence between scalar operations within a single vector CC instruction.

3.4.8 Memory Disambiguation and Store Coalescing

Similar to SIMD instructions, Compute Cache (CC) vector instructions require additional support in the processor core for memory ordering. We classify instructions in CC ISA into two types. CC-R type (`cc_cmp`, `cc_search`) only read from memory. The rest of the instructions are CC-RW type as they both read and write from memory. Under RMO memory model, CC-R can be executed out-of-order, whereas CC-RW behaves like a store. In the following discussion, we refer to CC-R as load, and CC-RW as store.

Conventional processor cores use a load-store queue (LSQ) to check for address conflicts between a load and the preceding uncommitted stores. As vector instructions can access more than a word, it is necessary to enhance the LSQ with the support for checking address ranges, instead of just one address. For this reason, we use a dedicated vector LSQ, where each entry has additional space to keep track of address ranges for the operands of a vector instruction.

Similar to LSQ, we also split the store buffer into two, one for scalar stores and another for vector stores. The vector store buffer supports address range checks (max 12 comparisons/entry). Our scalar store buffer permits coalescing. However, it is not possible to coalesce CC-RW instructions with any store, because their output is not known till they are performed in a cache. As the vector store buffer is non-coalescing, it is possible for the two store buffers to contain stores to the same location. If such a scenario is detected, the conflicting store is stalled until the preceding store is complete which ensures program order between stores to the same location. We augment the store buffer with a field which points to any successor store and a stall bit. The stall bit is reset when the predecessor store completes.

Data values are not forwarded from vector stores to any loads, or from any store to a vector load. Code segments where both vector and scalar operations access the same location within a short time span is likely to be rare. If such a code segment is frequently executed, the compiler can choose to not employ Compute Cache optimization.

3.4.9 Error Detection and Correction

Systems with strong reliability requirements employ Error Correction Codes (ECC) for caches. ECC protection for conventional and near-place operations are unaffected in our design. For cc_copy simply copying ECC from source to destination suffices. For cc_buz, ECC of zeroed blocks can be updated. For comparison and search, ECC check can be performed by comparing the ECCs of the source operands. An error is detected if data bits match, but the ECC bits don't, or vice versa.

For in-place *logical* operations (cc_and, cc_or, cc_xor, cc_clmul, and cc_not), it is challenging to perform the check and compute the ECC for the result. We propose two alternatives. One alternative is to read out the xor of the two operands and their ECCs, and check the integrity at the ECC logic unit ($ECC(A\ xor\ B) = ECC(A)\ xor\ ECC(B)$). This unit also computes the ECC of the result. Our sub-array design permits computing the xor operation alongside any logical operation. Although the logical operation is still done in-place, this method will incur extra data transfers to and from the ECC logic unit. Cache scrubbing during cache idle cycles [122] is a more attractive option. Since soft errors in caches are infrequent (0.7 to 7 errors/year [146]), periodic scrubbing can be effective while keeping performance and energy overheads low.

3.4.10 Near-Place Compute Caches

In the absence of operand locality, we propose to compute instructions "near" the cache. Our controller is provisioned with additional logic units (not arithmetic units) and registers to temporarily store the operands. The source operands are read from the cache sub-array

into the registers at the controller, and then computed results are written back to the cache. In-place computation has two benefits over near-place computation. First, it provides massive compute capability for almost no additional area overhead. For example, a 16 MB L3 with 512 sub-arrays allows 8KB of data to be computed in parallel. To support equivalent computational capability, we would need 128 vector ALUs, each of width 64-bytes. This is not a trivial overhead. We assume one vector logic unit per cache controller in our near-cache design. Second, in-place compute avoids data transfer over H-Tree wires. This reduces in-place compute latency (14 cycles) compared to near-cache (22 cycles). Also, 60%-80% of total cache read energy is due to H-Tree wire transfer (See Table 3.1), which is eliminated with in-cache computation. Nevertheless, near cache computing retains the other benefits of Compute Caches, by avoiding transferring data to the higher-level caches and the core.

3.5 Applications

Our Compute Cache design supports simple but common operations, which can be utilized to accelerate diverse set of data intensive applications.

Search and Compare Operations: Compare and search are common operations in many emerging applications, especially text processing. Intel recently added seven new instructions to the x86 SSE 4.2 vector support that efficiently perform character searches and comparison [15]. The Compute Cache architecture can significantly improve the efficiency of these instructions. Similar to specialized CAM accelerators [64], our search functionality can be utilized to speed up applications such as, search engines, decision tree training

and compression and encoding schemes.

Logical Operations: Compute Cache logical operations can speedup processing of commonly used bit manipulation primitives such as *bitmaps*. Bitmaps are used in graph and database indexing/query processing. Query processing on databases with bitmap indexing requires logical operation on large bitmaps. Compute Caches can also accelerate binary *bit matrix multiplication (BMM)* which has uses in numerous applications such as error correcting codes, cryptography, bioinformatics, and Fast Fourier Transform (FFT). Given its importance, it was implemented as a dedicated instruction in Cray supercomputers [8] and Intel processors provision a x86 carryless multiply ($clmul$) instruction to speed it. Inherent cache locality in matrix multiplication makes BMM suitable for Compute Caches. Further, our large vector operations can allow BMM to scale to large matrices.

Copy Operation: Prior research [126] makes a strong case for optimizing copy performance which is a common operation in many applications in system software and warehouse scale computing [75]. The operating system spends a considerable chunk of its time (more than 50%) copying bulk data [28]. For instance copying is necessary for frequently used system calls like fork, inter-process communication, virtual machine cloning and deduplication, file system and network management. Our copy operation can accelerate checkpointing, which has a wide range of uses, including fault tolerance and time-travel debugging. Finally, our copy primitive can also be employed in bulk zeroing which is an important primitive required for memory safety [154].

Configuration	8 core CMP
Processor	2.66 GHz out-of-order core, 48 entry LQ, 32 entry SQ
L1-I Cache	32KB, 4-way, 5 cycle access
L1-D Cache	32KB, 8-way, 5 cycle access
L2 Cache	inclusive, private, 256KB, 8-way,11 cycle access
L3 Cache	inclusive, shared, 8 NUCA slices, 2MB each, 16-way, 11 cycle + queuing delay
Interconnect	ring, 3 cycle hop latency, 256-bit link width
Coherence	directory based, MESI
Memory	120 cycle latency

Table 3.4: Simulator parameters

3.6 Evaluation

In this section we demonstrate the efficacy of Compute Caches (CC) using both micro-benchmark study and a suite of data-intensive applications.

3.6.1 Simulation Methodology

We model a multi-core processor using SniperSim [29], a Pin-based simulator per Table 3.4. We use McPAT [90] to model power consumption in both cores and caches.

3.6.2 Application Customization and Setup

In this section we describe how we redesigned applications in our study to utilize CC instructions.

WordCount: WordCount [157] reads a text file (10MB) and builds a dictionary of unique words and their frequency of appearance in the file. While the baseline does a binary search over the dictionary to check if a new word is found, we model the dictionary as alphabet indexed (first two letters of word) CAM (1KB each). As the dictionary is large (719KB) we perform search operations in L3 cache. CC search instruction returns a bit

vector indicating match/mismatch for multiple words and hence we also model additional mask instructions which report match/mismatch per word.

StringMatch: StringMatch [157] reads words from a text file (50MB), encrypts them and compares them to a list of encrypted keys. Encryption cannot be offloaded to cache, hence, encrypted words are present in L1-cache and we perform CC search in it. By replicating an encrypted key across all sub-arrays in L1, a single search instruction can compare it against multiple encrypted words. Similar to WordCount we also model mask instructions.

DB-BitMap: We also model FastBit [2] a bitmap index library. The input database index is created using data sets obtained from a real physics experiment, STAR [7]. A sample query performs logical OR or AND of large bitmap bins (several 100 KBs each). We modify the query to use cc_or operations (each processes 2KB of data). We measure average query processing time for a sample query mix running over uncompressed bitmap indexes.

BMM: Our optimized baseline BMM implementation (Section 3.5) uses blocking and x86 CLMUL instructions. Given the reuse of matrix we perform cc_clmul in L1-cache. We model 256×256 bit matrices.

Checkpointing: We model in-memory copy-on-write checkpointing support at page granularity for SPLASH-2 [147] benchmark suite (checkpointing interval of 100,000 application instructions).

3.6.3 Compute Sub-Array: Delay and Area Impact

Compute Caches have negligible impact on the baseline read/write accesses as we still support differential sensing. To get delay and energy estimates, we perform SPICE simulations on a 28nm SOI CMOS process based sub-array, using standard foundry 6T bit-cells. [1] A and/or/xor 64-byte in-place operation is $3\times$ longer as compared to single sub-array access while rest of CC operations are $2\times$ longer. In terms of energy, cmp/search/clmul are $1.5\times$, copy/buz/not are $2\times$, and the rest are $2.5\times$ baseline sub-array access. The area overhead is 8% for a sub-array of size 512×512 [2]. Note, our estimates account for technology variations and process, voltage and temperature changes. Further, these estimates are conservative when compared to measurements on silicon [73] in order to provision for robust margin against read disturbs and to account for circuit parameter variation across technology nodes.

We use the above parameters in conjunction with energy per cache access from McPAT to determine the energy of CC operations (Table 3.5). CC operations cost higher in lower-level caches as they employ larger sub-arrays. However, they do deliver higher savings (compared to baseline read/write(s) needed) as they have larger in-cache interconnect components. For search, we assume a write operation for key; this cost will get amortized over large searches.

[1] SRAM arrays we model are 6T cell based. Lower-level caches (L2/L3) are optimized for density and employ 6T-based arrays. However, L1-cache can employ 8T cell based designs. To support in-place operations in such a design, a differential read-disturb resilient 8T design [149] can be used.

[2] The optimal sub-array dimension for L3 and L2 caches we model are 512×512 and 128×512 bits respectively.

cache	write	read	cmp	copy	search	not	logic
L3	2852	2452	840	1340	3692	1340	1672
L2	1154	802	242	608	1396	608	704
L1	375	295	186	324	561	324	387

Table 3.5: Cache energy (pJ) per cache-block (64-byte)

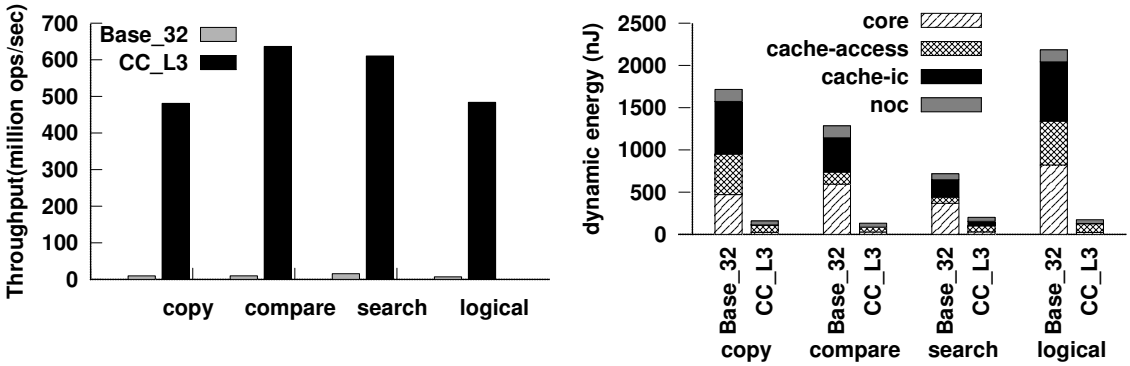


Figure 3.7: Benefit of CC for 4KB operand. a) Throughput b) Dynamic energy

3.6.4 Microbenchmark Study

To demonstrate the efficacy of Compute Caches we model four microbenchmarks: copy, compare, search and logical-or. We compare Compute Caches to a baseline($Base\_32$) which supports 32-byte SIMD loads and stores.

Figure 3.7 (a) depicts the throughput attained for different operations for operand size of 4KB. For this experiment, all operands are in L3 cache and the Compute Cache operation is

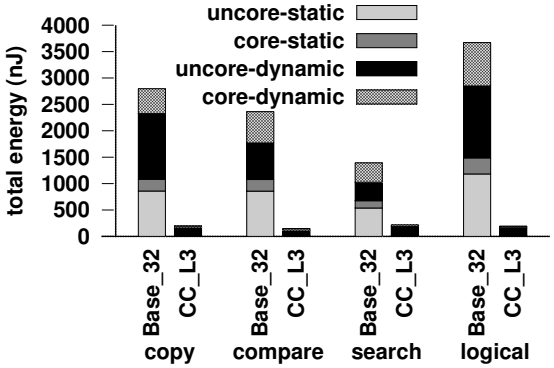


Figure 3.8: Total energy benefit of CC for 4KB operand.

performed therein. Among the operations, for baseline, search achieves highest throughput as it incurs single cache miss for the key and subsequent cache misses are only for data. Compute Cache accelerates throughput for all operations: $54\times$ over $Base\_32$ averaged across the four kernels. Our throughput improvement has two primary sources: massive data parallelism exposed in presence of independent sub-arrays to compute in, and latency reduction due to avoiding data movement to the core. For instance, for copy operation, data parallelism exposes $32\times$ and latency reduction exposes $1.55\times$ throughput improvement.

Figure 3.7 (b) depicts the dynamic energy consumed for operand size of 4KB. Dynamic energy depicted is broken down into core, cache data access (cache-access), cache interconnect (cache-ic) and network-on-chip (noc) components. We term data movement energy to be everything except the core component. Overall, CC provides dynamic energy savings of 90%, 89%, 71% and 92% for copy, compare, search and logical (OR) kernels relative to $Base\_32$. Large vector CC instructions help bring down core component of energy. Further, CC successfully eliminates all the components of data movement. Writes incurred due to key replication limit efficacy of search CC operation in bringing down L3 cache energy components. As data size to be searched increases, key replication overheads will get amortized increasing effectiveness of CC.

Figure 3.8 depicts total energy consumed broken down into static and dynamic components. Due to reduction in execution time, CC can significantly reduce static energy. Overall, averaged across the four kernels studied, CC provides 91% in total energy savings relative to $Base\_32$.

Near-place design:

In our analysis so far, we have assumed perfect operand locality i.e. all Compute Cache

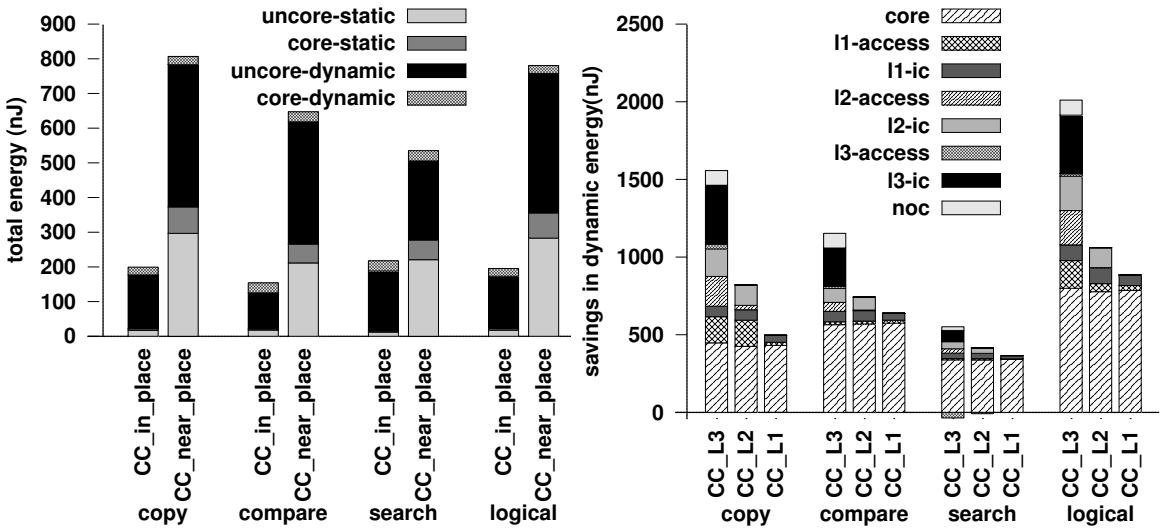


Figure 3.9: a) Total energy of in-place vs near place for 4KB operand b) Savings in dynamic energy for 4KB operand for different cache levels

operations are performed in-place. Figure 3.9 (a) depicts the total energy for near-place and in-place CC configurations. Recall that in-place computation enables far more parallelism than near-place and offers larger savings in terms of performance and hence total energy. For example, our L3-cache allows 8KB data to be operated in parallel. Near-place design would need 128 64-byte wide logical units to provide equivalent data parallelism. This is not a trivial overhead. As such, for 4KB operands, in-cache provides $3.6\times$ total energy savings and $16\times$ throughput improvement on average over near-place. Note however that, near-place can still offer considerable benefits over the baseline architecture.

Computing at different cache levels: We next evaluate the efficacy of Compute Caches when operands are present in different cache levels. Figure 3.9 (b) depicts the difference in dynamic energy between CC configurations and their corresponding $Base\_32$ configurations. As expected, the absolute savings are higher, when operands are in lower-level caches. However, we find that doing Compute Cache operations in L1 or L2 cache can also provide significant savings. As the number of CC instructions stays same regardless of

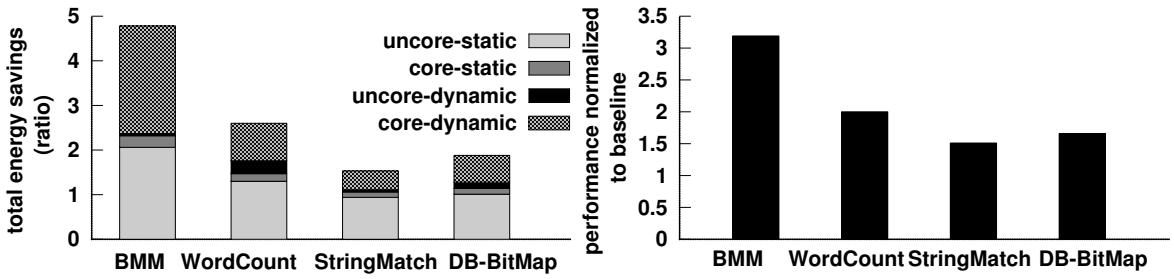


Figure 3.10: a) Total energy benefit b) Performance improvement of CC for applications

cache level, core energy savings is equal for all cache levels. Overall, CC provides savings of 95% and 93% for L1 and L2 caches respectively relative to $Base\_32$.

3.6.5 Application Benchmarks

In this section we study the benefits of Compute Caches for five applications. Figure 3.10 (b) shows the overall speedup of Compute Caches for four of these applications. We see a performance improvement of 2× for WordCount, 1.5× for StringMatch, 3.2× for BMM, and 1.6× for DB-BitMap. Figure 3.10 (a) shows ratio of total energy of CC to baseline processor with 32-byte SIMD units. We observe average energy savings of 2.7× across these applications. Majority of benefits come from three sources: data parallelism exposed by large vector operations, reduction in number of instructions and data movement.

For instance, recall that while baseline WordCount does a binary search over dictionary of unique words, Compute Cache does a CAM search using cc_search instructions. Superficially it may seem that binary search will outperform CAM search. However, we find that CC version has 87% fewer instructions by doing away with book keeping instructions of binary search. Further, our vector cc_search enables energy efficient CAM searches. These benefits are also evident in StringMatch, BMM and DB-BitMap (32%,

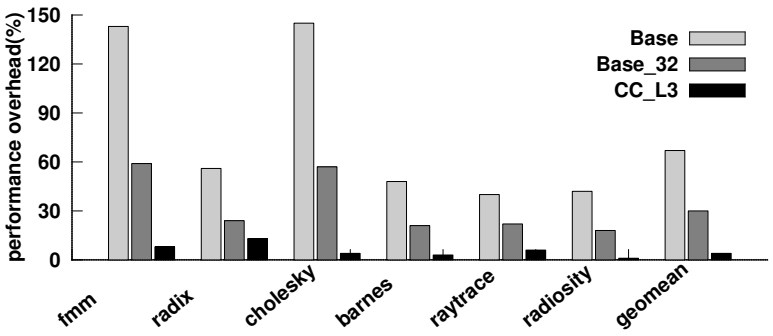


Figure 3.11: Performance overhead of CC for checkpointing

98% and 43% instruction reduction respectively). The massive data level parallelism we enable benefits data intensive range and join queries in DB-BitMap application. Recall that this benchmark performs many independent logical OR operations over large bitmap bins. Since these operations are independent, many of them can be issued in parallel.

Significant cache locality exhibited by these applications makes them highly suitable for Compute Caches. As cache accesses are cheaper than memory accesses, computation in cache is more profitable for data with high locality or reuse. The dictionary in WordCount has high locality. BMM has inherent locality due to the nature of matrix multiplication. In DB-BitMap, there is significant reuse within a query due to aggregation of results into a single bitmap bin, and there is potential reuse of bitmaps across queries. In StringMatch, locality comes due to repeated use of encrypted keys.

Figure 3.11 depicts the overall checkpointing overhead for SPLASH-2 applications as compared to baseline with no checkpointing. In absence of SIMD support, this overhead can be as high as 68% while in presence of it the average overhead is 30%. By further reducing instruction count and avoiding data movement, CC brings down this overhead to a mere 6%. CC successfully relegates checkpointing to cache, avoids data pollution of

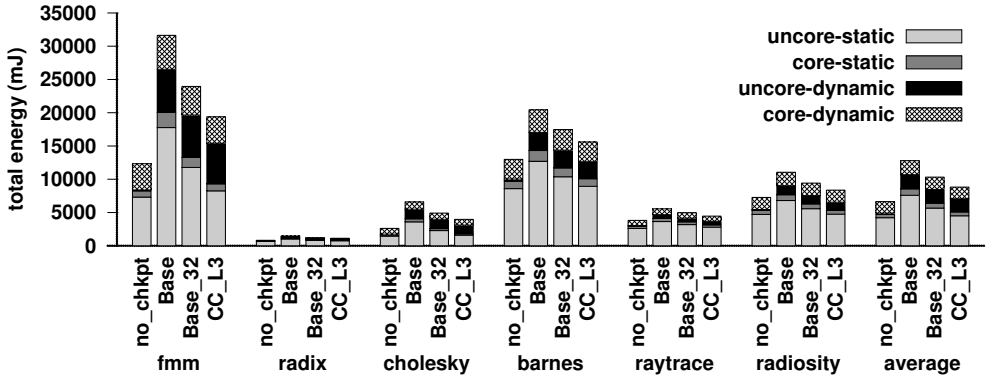


Figure 3.12: Total energy with and without checkpointing

higher level caches and relieves the processor of any checkpointing overhead. Figure 3.12 shows significant energy savings due to Compute Caches. Note that, for checkpointing, all operations are page-aligned and hence we achieve perfect operand locality.

3.7 Related Work

To our knowledge, Compute Caches is the first to make a case for off-loading computation to caches. Our SRAM design enables in-place computation. As this proposal focuses on reducing data movement and making it efficient, we compare and contrast prior work which addresses both of these aspects of data movement.

Near Memory Computing:

Concept of near memory computing or processing in memory (PIM) has been around for some time [58, 77, 86, 102, 106]. Recent advances with die-stacking which make integrating high-speed logic process technology with memory possible have renewed interest in near memory computing [12, 159, 53, 162, 109]. The key idea in these works is to place computation near main memory. Ahn *et al.* [22] proposed a locality aware PIM design, which opportunistically decides when computation should be offloaded to memory

and when it should be performed using host processor instructions. TCAM [64] enables in-memory associative computing using STT-MRAM technologies.

PIM is an effective approach for reducing data movement costs for applications that have no cache locality. However, for cache friendly applications, compute caches are a better choice. Ultimately, the locality characteristics of an application should guide in which level of memory hierarchy (registers, higher-level caches, lower-level caches, or memory) the computation must be performed.

Seshadri *et al.* [125] exploit existing DRAM operation to perform a bitwise AND/OR of two DRAM rows completely within DRAM. Our proposal complements [125] when applications have cache locality. For applications with cache locality, in-DRAM computation is energy inefficient compared to compute caches. Also, unlike our proposal, every logical operation in [125] first requires all operands in the logical operation to be copied to new DRAM rows because DRAM read is destructive. This incurs additional overheads. Overhead of such copies is justified only at large granularities. In-cache computation, however, can support much smaller granularities. Finally, while this work is similarly limited by operand locality, it does not discuss ways to address it but proposes copying of operands from one DRAM sub-array to another to perform in-place computation.

Processing in memory solutions like RowClone [126] exploit internal DRAM organization to perform fast copy operations and can be employed for in-memory checkpointing. However, offloading copying to memory requires flushing dirty data from higher level caches and will cause high latency cache misses for application data. In our proposal, by performing copy at various cache levels can avoid this cost.

Duarte et al [50] propose adding an indexing table near cache to accelerate cache line

copying. They neither provide in-place copy nor provide any other operation but copy.

Reducing data movement Several proposals from both hardware and software domain can reduce on-chip data movement. Non-uniform cache access(NUCA) organizations, in which large last level caches are decomposed into smaller slices have been widely studied [80, 66, 37, 45] to exploit variable wire lengths between cache slices and cores. Locality aware task mapping [54, 19, 156, 42] can also reduce data movement by placing frequently communicating tasks close to each other. Such techniques reduce data movement costs by moving data closer to computation cores that need them. Compute caches do the opposite – it moves compute near the data.

Optimizations to reduce cache misses [78, 148, 69, 128], conflicts [27, 79, 114], or placing data in locality tailored fashion [104, 45], energy efficient communication fabrics [99, 142, 44, 82, 139, 43, 18] can all in general reduce data movement over cache hierarchy. In addition to this, there is a rich literature which employs cache bypassing [70, 115, 74] by predicting reuse distances of data. Such techniques can complement our design by helping to effectively select the cache level to perform a CC operation.

Logic in memory arrays: Associative memories [103] perform XOR operations within the array, but have either much larger bit-cells or more complicated bit-cells [108] or non-CMOS bit-cells [64]. Also they have different peripheral circuit organization around the array. Our proposed solution does not modify the foundry provided, highly optimized, six transistor CMOS bit-cell and uses an organization compatible with conventional cache sub-arrays.

3.8 Discussion

While Section 3.7 compares Compute Caches to other contemporary works which focus on optimizing data movement, this section places Compute Caches in the larger context of available computation nodes. Programmers today have several architectures at their disposal (multi-cores, GPUs) to realize performance. Further, as discussed in Section 3.7, research proposals are also looking at using DRAM and emerging memories to perform computations. This raises an interesting question as to which computation node is best suitable for a given computation or in other words, how do these computation nodes compare against each other. We compare available and proposed computation nodes against Compute Caches along three dimensions: potential to reduce data movement costs, data-parallelism available and computation capability.

First, as this work demonstrates (Figure 3.3), general purpose cores expend a great deal of time and energy in data movement over cache hierarchies. As GPUs employ caches too, they also expend energy in data movement over them. In contrast, by enabling computation inside caches we can save this component of energy. Computation inside/near memory (DRAM and emerging memories like memristors) can avoid expending data movement energy over caches. However, in presence of cache locality for operands, computation in caches is far more profitable as each cache access is cheaper (delay and energy) as compared to a memory access.

Second, data parallelism in compute caches is orders of magnitude higher than SIMD units in CPUs, and even GPUs. For example, we can re-purpose 30 MB LLC in the server class Intel Xeon E5-2597 to support 983,040 bit-serial ALU slots. This compute capabil-

ity is significantly higher than aggregate SIMD width in Xeon (448 32-bit slots), or even Nvidia Titan Xp (3840 32-bit slots). While data parallelism in DRAM and emerging memories like memristors can be higher than present in caches (owing to their larger sizes), former does not support in-place operations due to destructive nature of DRAM reads necessitating copies and latter is a speculative technology, and is also significantly slower than SRAM.

Finally, the set of operations currently supported inside caches is limited as compared to that supported by a general purpose core or GPU or even memristors. However, as demonstrated in this work, Compute Cache enabled logical and copy operations can accelerate a wide variety of applications such as search engines, cryptography, bitmap based databases, bioinformatics and operating system primitives.

To conclude, nearly three-fourth of a server class processor die area today is devoted for caches. Even accelerators use large caches. Our work provides evidence that turning them into compute units can deliver significant performance and energy savings. As caches can be found in almost all modern processors, we envision compute caches to be a disruptive technology that can enhance commodity processors with large data-parallel accelerators for almost free of cost. CPU vendors (Intel, IBM, etc.) can thus continue to provide high-performance general-purpose processing, while enhancing them with a co-processor like capability to exploit massive data-parallelism. Such a processor design is particularly attractive for difficult-to-accelerate applications that frequently switch between sequential and data-parallel computation. As such, Compute Caches is a worthy addition to available computation nodes.

3.9 Conclusion

In this chapter we proposed the Compute Cache architecture which unlocks hitherto untapped computational capability present in on-chip caches by exploiting emerging SRAM circuit technology of bit-line computing. Using compute enabled caches, we can perform several simple operations in-place in cache over very-wide operands. This exposes massive data parallelism saving instruction processing, cache interconnect and intra-cache energy expenditure. We present solutions to several challenges exposed by such an architecture. We demonstrate the efficacy of our architecture using a suite of data intensive benchmarks and micro-benchmarks.

CHAPTER IV

InvisiMem: A Low-overhead Secure Processor

This chapter presents InvisiMem, a low-overhead secure processor design with strong defenses against memory side channels and efficient data freshness guarantees. Both memory side channel and data freshness are well studied problems and prior work has proposed solutions with various optimizations. However, in our work we look at these security vulnerabilities in new light of 3D stacked memory with integrated logic layer. Our proposal shows how in presence of such "smart" memories simpler and efficient solutions to these security vulnerabilities are possible.

4.1 Introduction

Cloud computing allows clients to outsource their computations to untrusted cloud service providers. Ensuring privacy of code and data on a computer physically owned and maintained by an untrusted party is challenging, as we must assume a powerful adversary. A malicious insider may even have physical access to the data-centers, making them vulnerable to physical attacks, such as probing the *memory bus* [10, 145].

A common solution is to reduce the attack surface by minimizing the trusted computing base (TCB) to a secure processor [9, 138] and a small portion of the client's application. Intel Software Guard Extensions (SGX) [97, 23] provides hardware primitives for this purpose. An SGX-enabled processor seeks to isolate code and data of private *enclave* functions in an application from the rest of the system, including its own public functions, system software, and hardware peripherals.

While the secure processor is trusted, the memory bus and the memory are not. Defending against memory bus side channel requires solutions to at least three problems: data and address confidentiality, data integrity and freshness, and timing channel. Solutions to these problems are expensive. For example, best known solution for address confidentiality is Oblivious RAM (ORAM) [60], which increases memory bandwidth consumption by ˜100X.

In this chapter, we present InvisiMem, a low-overhead secure processor that provides ORAM equivalent guarantees for the address side channel, ensures data integrity, freshness, and mitigates memory timing channel. InvisiMem is based on our observation that smart memories (memories with compute capability) with packetized interface (as opposed to the DDR interface) can be taken advantage of to design an ultra-low overhead secure processor. Recent advancements in 3D integration technology such as the Hybrid Memory Cube (HMC) [16] make it possible to stack DRAM layers on top of logic layers, and connect them using Through Silicon Vias (TSV). Unlike a memory bus that is exposed to an adversary, the TSVs pass completely through a silicon wafer, and therefore it is almost impossible to probe them without destroying the 3D package. Thus, there is no need for expensive mitigation solutions to protect the communication over the TSVs.

InvisiMem executes the private enclave functions in a SGX-like secure high-performance host processor connected to the smart memory via a memory bus using a packetized interface (Figure 4.1 (a)). The logic in smart memory is included in TCB and is used to implement cryptographic functions, while memory layers remain outside TCB.

Data and Address Confidentiality: A secure processor guarantees data confidentiality by encrypting data before sending it to memory. Memory address, however, is sent in plain-text on the bus, as required by the DRAM's DDR interface. By observing the memory addresses, an adversary can infer sensitive program inputs [151] and cryptographic keys [163]. Prior solutions tackle this leak using Oblivious RAM (ORAM) [60]; a cryptographic construction that obfuscates memory accesses to make them indistinguishable from a random access pattern. To do so, it issues several memory accesses for every normal access. In spite of significant recent advancements [56, 160] such ORAM-based solutions increase memory access latency considerably (˜20X) and incur huge performance overheads(˜4X). ORAM also has security limitations in that it does not help guard against memory timing channel [91].

InvisiMem's trusted compute capability in memory allows the processor to send the whole request packet, including the address, in an encrypted form. We observe, however, that *address encryption alone is insufficient to provide ORAM guarantees*. First, read requests or responses should appear indistinguishable from their write counterparts to an adversary snooping the memory bus to avoid leaking the memory access type. Second, on a read to a location, if the memory simply returns stored encrypted data at that location, an adversary can correlate it to the last write or previous reads to the same location. Solutions to address these problems are discussed.

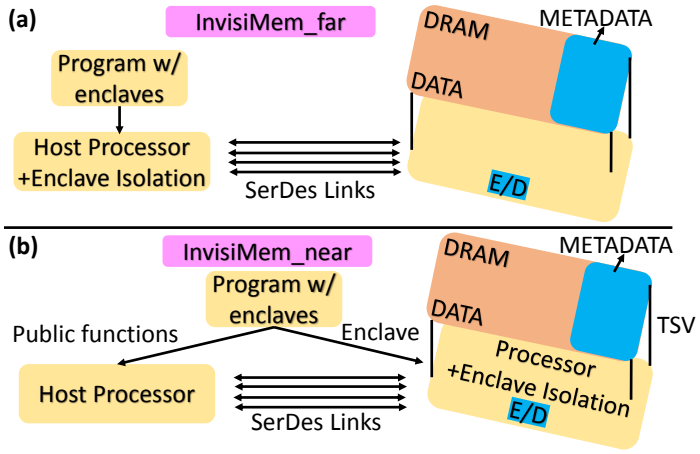


Figure 4.1: Smart memory based secure designs. a) InvisiMem_far b) InvisiMem_near

Freshness without Merkle trees: Guaranteeing data freshness requires that an adversary should not be able to rollback the state of a memory block by recording and replaying older packets (either by manipulating values in memory or while being transmitted over the bus). To defeat such replay attacks, a conventional secure processor maintains current versions of memory blocks using Merkle trees [117] and verifies that a read response returns the latest version. Merkle trees impose severe memory space and bandwidth requirements [63].

We observe that smart memory can provide freshness without requiring the Merkle tree construct. To do that we set up a secure communication channel between processor and memory using authenticated encryption. This is possible thanks to the compute capability of memory. Using authenticated encryption, we can prevent data manipulation in memory or over the memory bus as both parties can ensure that requests/responses are originating from the authenticated counterpart and that they are fresh.

Timing Channel: Smart memory also enables an efficient solution for solving one type of memory timing channel: memory access and response times seen on the memory bus.

InvisiMem sends constant rate heart-beat packets between the processor and smart memory in both directions. When an actual request or response is available, the next heart-beat packet's slot is used or else a dummy packet is transmitted. This is viable for systems with smart memory (HMC-like) for several reasons. First, the interface employed in such systems sends synchronization packets periodically even when there is no communication [12]. Hence, the energy overhead of turning them into heart-beat packets is relatively low. Second, unlike traditional memory systems, smart memory allows dummy requests to be ignored, lowering their energy overhead. Third, compute capability in smart memory allows responses from memory to be sent at a constant rate (unlike traditional memory systems). This helps hide variations in access times to different locations.

The above solution naturally supports a system with multiple memory modules connected to a processor. Sending constant rate heart-beat packets between every pair of communicating nodes hides all access patterns, and thereby also hides addresses accessed. We also discuss some precautions needed to support such systems.

Optimizations: Encrypting and decrypting packets constitute the majority of the performance overhead in InvisiMem. Specifically, computing OTPs (one-time pads) using AES incur the highest latency. We take these operations out of the critical path of a memory access by *precomputing OTPs* before a request/response is sent or received. We also investigate various designs for efficiently storing and retrieving meta-data (for encryption and integrity checks) which exploit smart memory characteristics like vault-level parallelism.

With these optimizations, InvisiMem incurs 14.21% performance overhead, 53.03% energy overhead and 37.5% memory space overhead compared to an Intel Xeon-like pro-

cessor.

The logic layer in smart memory has sufficient area and power budget (nearly 55W [51]) for a low-power processor. Executing enclaves in this core can hide all the communication between the core and memory, and thereby eliminate memory bus side channel. The trade-off, however, is that its compute capability may not match that of a high-performance core. We study this using a variant of our design, named InvisiMem_near (Figure 4.1 (b)).

Remote attestation and Key Management: InvisiMem expands TCB to include logic layer stacked with memory. Note that DRAM layers in memory are still outside TCB. Almost all secure processors such as Intel SGX, including all ORAM based designs, rely on public key infrastructure (PKI). Similarly, we propose that smart memory vendors also support PKI for trusted logic in memory. Using the public key of the smart memory, we show that a secure host processor can easily establish a secure communication channel with the smart memory's logic.

4.2 Motivation and Background

In this section we briefly describe hardware support for secure containers (enclaves), which we assume in our work. We also present a threat model and discuss prior defenses for memory bus side channel. Finally, we provide a brief background on 3D stacked memory, which we use in our system.

Channel	Leak/Vulnerability	Freecursive ORAM [56]	Ghostrider [91]	InvisiMem_far	InvisiMem_-near
Passive Memory Bus Probe	Data	Data encryption	Data encryption	Data encryption	Eliminate memory bus channel
	Address	ORAM	ORAM	Whole packet encryption + Double data and timestamp encryption	
	Access type (R/W)	ORAM	ORAM	Same packet size for read/write	
	Trace length	with [57], yes	Deterministic execution	constant rate requests/responses	
	Access time	with [57], yes	Deterministic execution	constant rate requests/responses	
Active Memory Bus Probe	Data	Data encryption	Data encryption	Enclave checks + Authenticated Encryption (HMAC)	Enclave checks + Authenticated Encryption (HMAC)
	Data corruption	HMAC	no		
	Replay attack	HMAC + access count + position data checks	no		
	Write set	ORAM	ORAM		
Cold Boot	Data	Data encryption	Data encryption	Data encryption	Data encryption
System software	Execution time	no	Deterministic execution	no	no

Table 4.1: Comparison of InvisiMem to ORAM-based defenses. Smart memory enables more efficient and simpler solutions.

4.2.1 Enclaves for Isolation

Intel SGX [23, 97] is the latest hardware support for building trusted computing systems. It provides capability for isolating the execution of an enclave from the rest of the system, including the public functions of the application, system software, and other hardware peripherals.

An enclave is a secure container that contains private data and the code that operates on it. An application is responsible for specifying enclaves and invoking them. When an enclave is invoked through special CPU instructions, the untrusted system software loads the enclave contents to the portion of the protected memory allocated for the enclave's execution. The secure processor computes the enclave's measurement hash over initial data and code, which the remote client uses for software attestation. Thereafter, the enclave is executed in a protected mode, where the hardware checks ensure that every memory access to protected memory is from its enclave.

4.2.2 Threat Model

We assume a secure processor that supports isolated execution of enclaves (e.g. Intel SGX [23, 97]). We assume that an adversary cannot observe the communication between the layers in smart memory and that its logic layer is secure.

We assume a powerful adversary that can compromise the operating system and use OS privileges to compromise the confidentiality and integrity of applications. This adversary also has physical access to the computers running client computations. Thus, he can probe the off-chip memory bus to observe (and modify) the communication between the secure processor and the memory, including the event times. We assume that DRAM die is untrusted, as the adversary may have the capability to scan DRAM contents through cold (re)boot attacks [65] or corrupt state using Row-Hammer attacks [84].

We assume that the execution of a private enclave function and its data in the processor (registers, caches, on-chip interconnect, performance counters) is secure and isolated from other computation. Several prior studies have discussed solutions for ensuring this property in a multi-core processor with shared hardware structures [40, 127, 143, 33]. We also assume prior solutions for mitigating page-fault side-channel [40, 132] in enclaved systems. Power [85], thermal [110], program execution time [158] side-channels, and leaks via communication patterns over the network [100, 124], have been addressed in prior work and are outside the scope of this work.

4.2.3 Memory Bus Side Channel and Cold Boot Defenses

Table 4.1 compares various leaks through memory bus side channel and cold boot attack that secure processors must protect. It describes the solutions used in two of the recent ORAM-based work, Freecursive ORAM [56] and Ghostrider [91]. While more recent ORAM-based work exists [160], we consider Freecursive ORAM [56] as it proposes a ORAM-based defense optimized for providing data integrity and freshness guarantees as well.

In most trusted computing systems such as SGX, all the hardware components outside the secure processor are untrusted, including the memory and the memory bus. To ensure confidentiality, they use randomized encryption to encrypt the data before writing to memory, and decrypt it when it is read back. This protects sensitive data from leaking directly through memory bus probes and cold boot attacks. However, an adversary can still observe addresses and access types (read or write) by passively probing the bus.

To protect confidentiality of addresses (also, access types and write sets), prior solutions employ expensive ORAM [60] construct. To obfuscate the address pattern, depending on the memory size, an ORAM access may require one to two orders of more memory accesses compared to a normal DRAM access. Recent hardware innovations such as Freecursive ORAM have made significant improvements to bring down the performance cost to about 4X [56], though at a significant increase in hardware complexity, on-chip (80KB) and off-chip (more than 2X) space overhead.

ORAM does not prevent memory access time and total number of memory accesses from leaking. This is provided by the memory-trace obliviousness (MTO) guarantee pro-

vided by Ghostrider [91] which ensures that the program execution (instruction trace, time) is independent of program sensitive inputs. This requires a deterministic compiler, hardware which prohibits most commonly used optimizations (caches, instruction re-ordering etc) and also imposes non-trivial constraints on the program (e.g. sensitive input-independent loop guards). As a result, it incurs nearly 6X performance overhead. compared to a baseline with a single-issue, in-order processor.

An adversary can corrupt data in memory and violate data integrity. A replay attack is also possible, where an adversary manages to rollback the state of a memory block by replaying an older write message. To provide data integrity, secure processors typically create and store hash message authentication code (HMAC) along with data in memory. On a read, HMAC can be used to detect data integrity violation. Replay attacks are thwarted by including a version number during the HMAC creation. The processor tracks the current version of the memory state using on-chip storage [152, 56], and uses it to ensure that a read returns the latest version. Both these guarantees incur additional performance and space overhead, and complexity.

With 3D smart memory, and by increasing the trusted computing base to include its logic layer, we can reduce the complexity of these problems, and realize low-overhead security solutions. *We seek to guarantee memory-trace obliviousness (MTO) property (except not protecting program execution time from leaking) along with data integrity and freshness guarantees.*

4.2.4 Smart Memory

3D integration has led to the rise of 3D-DRAM devices such as the Hybrid Memory Cube (HMC) [16]. A typical 3D-DRAM consists of several layers of DRAM dies stacked on top of each other, with a logic layer at the bottom, all internally connected using Through Silicon Vias (TSVs) [12]. The layers are partitioned vertically into vaults (each with several DRAM banks) which can be accessed in parallel. HMC device is connected to the processor via SERDES links. Unlike traditional DRAM's DDR interface with low-level commands, HMC device is exposed via a more flexible packet interface.

Recent HMC device has a capacity of 2GB and can provide maximum memory bandwidth of 160GB/s [16]. While the logic layer in current devices contains circuits for interfacing with the vaults (memory controllers), it has sufficient area and thermal power budget (55W [51]) to include fairly sophisticated computational units, such as a low-power processor and/or cryptographic units.

In 2.5D stacking, the memory and the processor can be interconnected through metal layers within a silicon interposer [47]. These metal tracks are etched using the same processes as the tracks on the silicon chips, and hence they are orders of magnitude smaller than the tracks on a conventional memory bus. It is therefore reasonable to assume that an adversary will be unable to tap the communication between the processor and memory in 2.5D system, providing similar security properties as 3D. Also, since logic is not stacked at the bottom of the memory layers, the thermal power budget would allow it to support a high-performance core.

4.3 InvisiMem Design

InvisiMem builds upon enclave support similar to Intel SGX or Sanctum [40] for isolation. As memory layers in smart memory are untrusted (cold-boot attacks, Section 4.2.2), we store encrypted data in memory using randomized encryption, similar to SGX.

We first discuss InvisiMem_far, which executes the enclave in a secure high performance processor (Figure 4.1 a)). Later, we discuss a more optimized design, InvisiMem_near, which executes the enclave within smart memory's logic.

We start by discussing smart memory advantages which help design low-overhead defenses for memory bus side channel and also lead to an efficient solution to guarantee freshness. We also discuss performance optimizations that we employ and efficient storage of meta-data in smart memory.

4.3.1 Advantages of Smart Memory

Compute capability in memory allows whole memory packets to be encrypted and decrypted. Also, it makes it possible to generate dummy responses, and discard dummy requests.

In traditional DRAM systems, on-chip memory controller issues low-level DDR standard compliant commands to interact with the off-chip DRAM modules. In contrast, a smart memory has a packetized interface. The logic layer in smart memory decodes command packets from processor and internally routes them to the memory controller associated with every vault. The memory controller then communicates with the DRAM memory in its vault through DDR commands. Smart memory's packetized design allows us to seam-

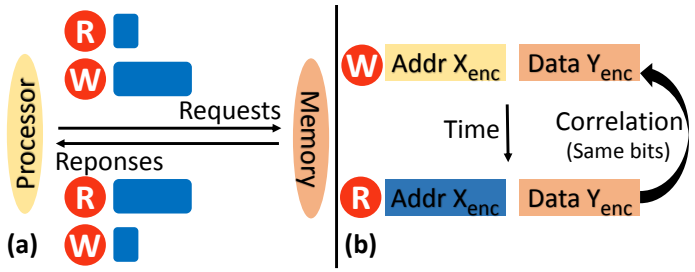


Figure 4.2: Symmetric encryption of addresses is not enough. (a) Distinguishing reads from writes (b) Correlation attack

lessly extend its packet processing logic with security functionality, without modifying the DDR standard, which is harder.

Unlike a memory bus, the TSVs that connect the logic layer and the DRAM memory pass entirely through silicon. It is almost impossible for an adversary to launch a physical attack by probing the TSVs without destroying the 3D package.

4.3.2 Protecting Memory Address and Type

In InvisiMem, secure processor encrypts and sends the whole packet, including data, address, access type (read or write) using randomized encryption. This is possible only because smart memory is capable of decrypting addresses. Randomized encryption makes it hard for an adversary to correlate messages that carry the same address. However, encrypting address alone is not enough to ensure ORAM properties.

First, an adversary can correlate a read to a location with an earlier write to the same location by simply comparing the encrypted data (or timestamp used to encrypt it) as depicted in Figure 4.2 (b). To solve this problem, while responding to a read request, the smart memory double encrypts an already encrypted data and its timestamp, before sending a response.

Second, the communication between the processor and the memory in an insecure design is noticeably different for reads and writes (Figure 4.2 (a)). A read request and a write response do not carry data, while a write request and a read response do. Thus, an attacker could infer whether an access is a read or a write. We eliminate this leak by ensuring equal packet sizes for both read and write request/responses by adding a dummy block to read request and write response.

These solutions are sufficient to provide guarantees equivalent to ORAM. However, they are not sufficient to prevent the number of memory accesses and their access times from leaking (ORAM leaks these too). Furthermore, response times may vary depending on the memory location accessed. We address these timing channel problems in Section 4.3.4.

4.3.3 Guaranteeing Data Integrity and Freshness

Our threat model assumes that DRAM layers are untrusted and therefore stored data can be corrupted (e.g. Row-hammer attacks [84]). An adversary may also corrupt data communication on the bus through active probing. Creating and storing a hash message authentication code (HMAC) with data on a write, and checking the code on a read can solve these issues.

Guaranteeing freshness, however, requires more extensive support in conventional hardware. In a replay attack, an adversary manages to rollback the state of a memory block by replaying an older data. Replay attacks can be prevented by including a version number during the HMAC creation. The processor must securely track the current version of the memory state [152, 56], and use it to check if a read is returning the latest version. But these

data integrity checks incur additional performance and space overhead, and complexity.

InvisiMem_far uses authenticated encryption [96] to guarantee freshness. Authenticated encryption ensures integrity and freshness of data sent over the untrusted memory bus. Using authenticated encryption, the sender (processor or memory) generates and sends an authentication tag over the encrypted packet sent to the receiver. The receiver uses this tag to check if the packet is the latest message from the trusted sender.

Authenticated encryption [96] uses an one-time pad (encryption of monotonically increasing counter) to generate encrypted data over which the authentication tag is then generated. As such, on a message replay, receiver's regenerated tag (using latest one-time pad) will not match the received tag (replayed) causing authentication failure. Note that when memory responds, it generates this tag over double encrypted data and timestamp (encrypted) (Section 4.3.2). Unlike prior designs [152], we avoid significant hardware state and memory space needed to track and check the versions of memory blocks.

The secure logic in memory performs the integrity checks only for accesses to protected memory range reserved for enclaves. It relies on the secure host processor to perform the necessary enclave checks to ensure that the accesses to enclave locations are from the enclave that owns them.

4.3.4 Mitigating Memory Bus Timing Channel

Memory access times observed on the memory bus can leak the program paths taken in an execution [57]. Memory response times to requests can also leak sensitive information. For example, reads with row-buffer locality will have significantly lower response latency than reads to different rows (Figure 4.3).

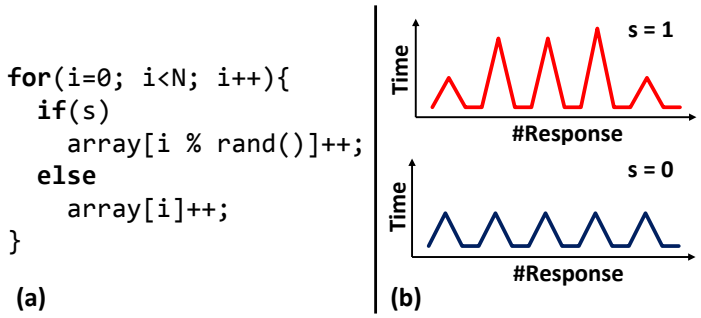


Figure 4.3: Time taken to respond by memory can leak sensitive inputs.

To solve both these leaks, the processor and the memory send heart-beat packets at a constant rate to each other. When there is a real packet to send, the sender transmits it at the next available slot. In the absence of a real packet, a dummy packet is sent, which is ignored by the receiver. This design trivially eliminates the two leaks noted above. Smart memory's capability to generate packets at a constant rate makes this design feasible. In a conventional memory system, as only the processor can send requests at a chosen rate, variations in response times noted above (Figure 4.3) are hard to mask.

We also experimented with a dynamic scheme that adjusted the packet rate according to application's memory access characteristics [57], but we did not find any significant performance or energy benefit in the context of a smart memory based design (Section 4.5.4).

We believe this is partly due to unique characteristics of smart memories. Smart memory is different from traditional memory systems modeled in prior work [57] in two aspects. First, smart memory can ignore dummy requests. Second, idle energy expended in smart memory is very high compared to a traditional DDR interface, as SerDes links in packetized interface require null packets to be sent at a constant rate for synchronization [12]. Link energy to transmit a null packet is about 75% of energy to send an actual packet [81].

Further, the energy expended in encrypting/decrypting packets does not constitute a

significant fraction of the total system energy, even while operating at a packet rate that is high enough for the most memory intensive programs we studied.

As a dynamic scheme's security guarantee is also weaker than a static rate, we chose the latter. Instead of choosing a constant packet rate for all applications, we could select a rate for each application using profiling or user input, without sacrificing security properties. Though we did not find a significant benefit for this approach, it may be useful for very memory intensive applications needing a higher packet rate than what we chose.

While outside the scope of this work, an attacker capable of measuring power side channel can distinguish real and dummy requests in InvisiMem, as the smart memory ignores dummy requests. If this is a concern, instead of ignoring a dummy request, we can issue an access to a random location.

4.3.5 Performance: OTP Pre-computation

One-time pad (OTP) generation, which uses an AES encryption, is the most time consuming portion of GCM [96] which we use for authenticated encryption. We take it off the critical path of a request or a response by pre-computing it.

An OTP is generated from a timestamp counter and a private key. A timestamp counter's state is shared between the processor and the smart memory. We enable sharing by initializing the respective timestamp counters in both processor and memory at the start of a program's execution to the same value. Thereafter, the sender and the receiver synchronously increment the counter on sending or receiving a packet.

Synchronous timestamp counters avoid sending the timestamps along with each packet. More importantly, the sender or the receiver can pre-compute an OTP even before a packet

is ready to be encrypted or decrypted. The only case where this is not possible is when decrypting a read response as the timestamp stored with the data must be first recovered to compute the OTP and then decrypt the data using it. See Section 4.4.2 and Figure 4.5 for more details.

Synchronous timestamp counters are feasible as the communication network is point-to-point between the processor and memory, and is generally lossless. If a communication link is unreliable, then timestamp counters can lose synchronization when a packet is lost. Unreliable networks typically deal with lost messages by tagging packets with sequence numbers, and re-synchronize when a packet loss is detected. Similar techniques can be used to track lost packets in our system.

4.3.6 Space: Meta-Data in Smart Memory

We consider two design alternatives for storing meta-data (timestamp and tag) with their data. In the fragmented design, data of a cache block is stored along with its meta-data in memory. This design has relatively lower complexity as the memory controller can fetch both data and its meta-data using a single request. However, storing meta-data with data consumes two cache blocks worth of space, even though meta-data is smaller than a cache data block.

In the non-fragmented design, we store data and meta-data at non-contiguous locations. This allows meta-data of multiple data blocks to be compactly packed together with less wastage of memory. However, this requires two requests per data access. We reduce any potential performance overhead due to the serialization of these requests by exploiting vault-level parallelism (Section 4.2.4). To achieve this, we map addresses to physical

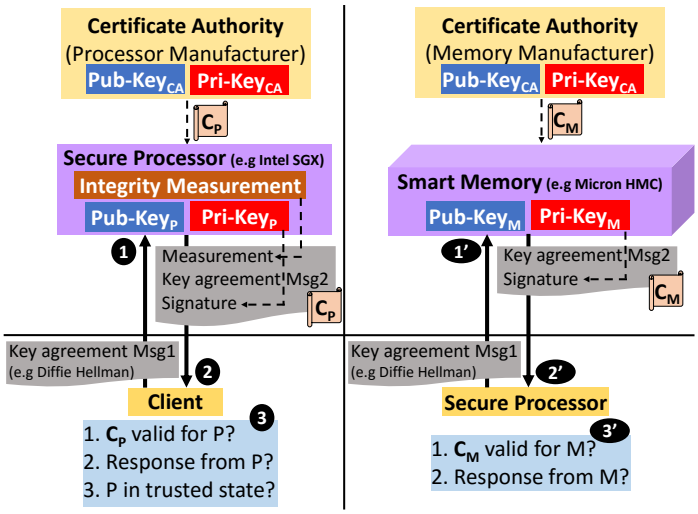


Figure 4.4: Existing client-host remote attestation and key exchange (left). Smart memory authentication and key exchange under InvisiMem (right).

locations such that data and its meta-data are always stored in different vaults (note however that in a multiple module system, data and it's associated metadata are present in a single module). Furthermore, as adjacent data blocks may be accessed in close succession, our mapping ensures that data and meta-data of spatially adjacent data blocks in the address space are stored in different vaults. See Section 4.4.4 for details. With these performance optimizations, non-fragmented design incurs only a negligible performance overhead compared to the fragmented design, but has better space utilization (91.66% compared to 68.75%).

4.3.7 Remote Attestation and Key Exchange

We now discuss a simplified client-processor remote attestation protocol [9, 23, 39], and how we could adapt it to set up a secure communication channel between the processor and memory in InvisiMem.

Remote attestation is a process by which a remote host proves to a client that it is run-

ning trusted software on trusted hardware. On successful conclusion of remote attestation, the client shares its sensitive data (e.g., private keys) with the host before commencing computation.

A secure processor manufacturer endows each secure processor with a unique public-private key pair. It also serves as a certificate authority that provides a *certificate* that binds the processor's identity to its public key. In addition, a secure processor has support for integrity measurement (a hash of code, data, and system state).

The client aims to attest a remote processor and setup a shared session key to communicate sensitive data with it. To do that, it sends a key agreement message (❶) to the remote host [48]. The processor uses this to generate a response key agreement message. This along with its integrity measurement is signed with the processor's private key. The signed message is sent along with the processor's certificate issued by its manufacturer (❷) to the client. The client (❸), first verifies that the certificate is valid using manufacturer's public key. Then, using the processor's public key in the certificate, it verifies that the received message is indeed from the processor. It further checks if the measurement value is as expected. If it is, then the client uses the key agreement message received to compute the shared session key for further secure communication.

In InvisiMem, the secure processor and the client use the conventional remote attestation protocol described above. Secure processor uses a similar protocol to authenticate and exchange keys with its secure memory. The only difference is that the secure memory does not need integrity measurement support. To support this, just like processor manufacturers, we propose that memory manufacturers endow smart memory modules with public-private key pair, and serve as its certification authorities.

To support the above protocol, we assume that smart memory's logic can support asymmetric encryption. Given its area and thermal budgets (Section 4.2.4), it should be able to support asymmetric encryption which is implemented even in smart-cards [68] with much lesser resources.

4.3.7.1 Security Considerations

Remote attestation protocol discussed above is immune to man-in-the middle (MITM) attacks. The second check in the last step of the protocol described above (❸) ensures that the response received is indeed from the trusted entity. Private keys stored in secure processor and secure memory are tamper-proof. But to manage scenarios where vulnerabilities are discovered after deployment, certificate authorities can maintain certificate revocation lists. Alternatively, certificates can be associated with expiration dates. None of these problems and solutions are unique to processor-memory authentication and key exchange described above, as they exist in client-processor remote attestation as well.

4.3.8 Key and Timestamp Management

Storage: Keys (data and address) and timestamps we employ are stored in special registers at memory controllers at both side. To ensure process isolation, each process has a different key. But we need only as many special registers for keys as there are processor cores, as the number of active processes cannot be more than that. A timestamp can be shared amongst processes. Security is guaranteed as long as a given timestamp is never reused for the same key, which is ensured by incrementing it each time it is used. Note that techniques that tackle timestamp overflow [152] can be adapted in our system.

Multiple Memory Modules: Systems where a secure processor is connected to multiple memory modules are possible [81]. Therein, the processor can setup a secure channel with each memory module. This will require a timestamp per memory module. Furthermore, to ensure that the same timestamp-key pair is not reused, we statically partition timestamp ranges amongst memory modules.

A security vulnerability in such a system is that an adversary can gain some information about an address accessed simply by observing which module is accessed. Fortunately, our timing channel solution (Section 4.3.4) addresses this problem.

4.3.9 Near InvisiMem

As noted in Section 4.2.4, the logic layer integrated with memory could support low-power (3D) or even high-performance cores (2.5D). InvisiMem_near exploits this opportunity by assuming that the secure host processor is within smart memory's logic layer. Since the TSVs are secure, it obviates the need for protecting communication between the core and memory. However, we conservatively exclude the DRAM memory layers from TCB (Section 4.2.2). Therefore, data is still stored in encrypted form in memory, and its integrity is checked by storing HMAC tags with data and checking them on read.

The memory controller bounces any access from an external device, including the host processor, by checking if it falls within the range of protected memory region dedicated for enclaves. Apart from these measures and support for enclave checks, all of which are already supported in commodity secure processors such as Intel SGX, InvisiMem_near requires little else support to provide the guarantees we seek. Note that this design does not need additional support to guarantee freshness or prevent memory timing channel leaks

(Section 4.3.4).

4.4 Implementation

This section describes hardware support for cryptographic primitives, and details how OTP pre-computation helps reduce the latency of encryption/decryption in a read/write operation, and how meta-data is stored in 3D memory efficiently.

4.4.1 Hardware Support for Cryptographic Primitives

4.4.1.1 Authenticated Encryption

We use Galois/Counter operation mode (GCM) [96] with AES for authenticated encryption. GCM operates on 128-bit blocks. Therefore, a single cache block (64 bytes) is broken into 4 blocks of plain-text. One Time Pad (OTP) is generated by using AES encryption on a counter along with a 128-bit encryption key. OTP is then XORed with a plain-text to generate its cipher-text. The counter used to generated OTP is incremented for every block that is processed to provide randomized encryption [116]. For authentication, GCM employs a GHASH function [96], which creates hash of a message ciphertext using a secret 128-bit hash key (H) derived from the encryption key. The output is an authentication tag, which is regenerated at the receiver to verify data integrity.

4.4.1.2 Metadata: Timestamps and Keys

InvisiMem_far uses three symmetric keys: address (*Ka*), data (*Kd*), and data double encryption (*Kd-de*). The data double encryption key is used to double encrypt encrypted

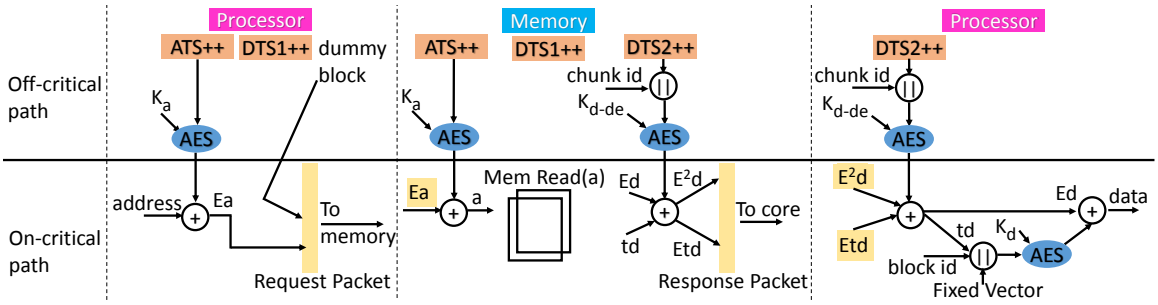


Figure 4.5: InvisiMem_far Security Protocol for Read. `td`: timestamp stored with data.

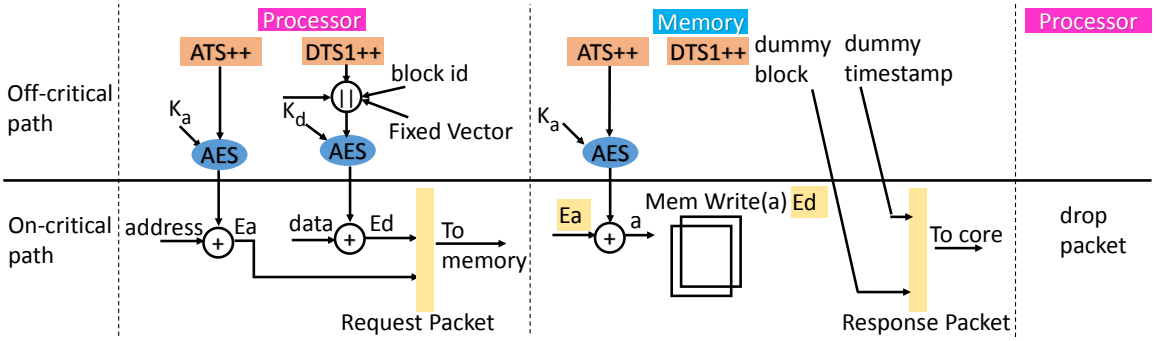


Figure 4.6: InvisiMem_far Security Protocol for Write.

data and timestamp to defend against correlation attack (Section 4.3.2).

We use three timestamps. 128-bit address timestamp (ATS) is used for generating address OTP. Address key (*Ka*) and ATS are used to encrypt packet header and tail, which includes the address, command, etc. For brevity, we simply refer to these in terms of encrypting/decrypting addresses in this section.

Smaller 64-bit data timestamp (DTS1) is used for encrypting 64-byte cache block data as follows. The cache block is broken into four 128-bit blocks. Timestamp for each block is produced by concatenating 64-bit timestamp (DTS1) with a 62-bit fixed vector (FV) and a two bit block-id representing it's relative position in the cache block. Since the timestamp for a cache block has to be stored along with data in memory, using a smaller 64-bit timestamp helps save space. For double-encryption of data and its timestamp, we use a 125-bit timestamp DTS2 concatenated with 3-bit chunk-id while generating the OTP.

This timestamp is never stored in memory.

4.4.1.3 Augmented Memory Controller

Smart memory has memory controller/s (MMC) in the logic layer which communicate with the integrated memory controller (PMC) in the processor. We augment both PMC and MMC to perform authenticated encryption (Figure 4.7 (a)). This requires registers for timestamps and keys mentioned in Section 4.4.1.2. PMC and MMC have three AES and four Galois Field multipliers (GF-M) [123] each.

4.4.2 InvisiMem_far Security Protocol

Figures 4.5 and 4.6 depict the steps involved in InvisiMem_far on a read and write respectively. We classify all the actions into either ”off-critical path” or ”on-critical path” of a read or a write access. For simplicity, we only depict the encryption part of GCM and not authentication tag generation which can be partly overlapped with encryption/decryption. We also ignore our timing channel solution, which simply requires that once a packet is ready it is sent at the next available slot.

In Figure 4.5, PMC encrypts an address for read request. Using ATS, we can pre-compute the OTP required for address encryption, leaving only an XOR operation on the critical path. Request packet for a read includes the encrypted address and dummy encrypted data block. The latter is added to the request packet to make it impossible for an attacker to differentiate a read request from a write request. On receipt of a request, MMC decrypts the address; again with a pre-computed OTP and issues a read to DRAM. On receiving a response from DRAM, MMC encrypts data and its associated timestamp

using pre-computed OTP generated from DTS2 (double encryption) and sends it to PMC. Double encryption is done to guard against correlation attack by observing encrypted data or timestamp. On receipt of response, PMC first decrypts data and timestamp using OTP pre-computed from DTS2. Then uses the decrypted timestamp to again decrypt the data, which is the expensive step in our protocol.

For a write (Figure 4.6), PMC encrypts data and address; while MMC decrypts address both using pre-computed OTPs. Thus, only XOR operations are on the critical path.

For authentication, a Galois Field multiplication and an XOR operation are also needed per ciphertext (Section 4.4.1.1). Read/write requests/responses require address and either data or dummy data tag generation on both PMC and MMC side. Dummy data tag generation can be avoided on receiving side. For a read response, MMC first checks the tag read from DRAM and generates another tag on double encrypted data for transmission. We can overlap some of these operations with data (and address) encryption/decryption.

4.4.3 InvisiMem_near Security Protocol

The protocol for near-memory secure processor involves data encryption and authentication tag generation on a write. A read requires data decryption, and authentication tag generation and check. While authentication delay is added to critical path for a write request, a read response overlaps it with data decryption.

4.4.4 Storing Meta-Data in Smart Memory

Meta-data for a cache block consists of 64-bit timestamp value (DTS1) and an integrity tag. Section 4.3.6 described two designs for storing this meta-data: fragmented and non-

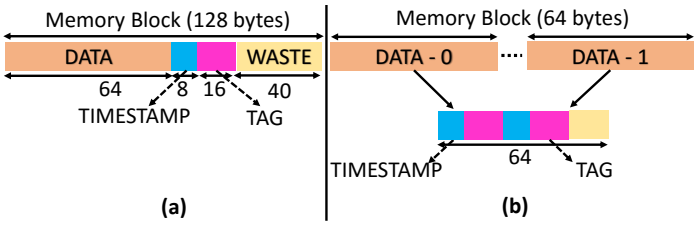


Figure 4.7: (a) Fragmented Design (b) Non-fragmented Design

fragmented.

Figure 4.7 (a) depicts fragmented layout. Storing data and its metadata together requires 88 bytes (64 data, 8 timestamp, 16 tag). HMC Specification [12] mandates that memory block sizes can be 32/64/128/256 bytes. Therefore, in the fragmented layout, 88-byte data block and its meta-data consumes 128-bytes, resulting in effective memory utilization of 68.75%.

Figure 4.7(b) depicts non-fragmented layout, where meta-data and data are not stored together. A 64 byte block can store meta-data for two data blocks (2 timestamps and 2 tags). This leads to a far better memory utilization of 91.66%. To exploit vault-level parallelism, our data mapper places data block and its meta-data in different vaults, so that they can be accessed in parallel. We also take care that meta-data of spatially adjacent data blocks are mapped to different meta-data blocks. Memory waits for both data and metadata before responding to a request from PMC.

4.5 Evaluation

4.5.1 Methodology

We study 22 benchmarks from SPEC 2006 [67] suite with reference inputs. We use the Simpoint [129] methodology with interval size of 100 million instructions to choose

Configuration	4 cores, commit width 4, 72 entry LQ, 42 entry SQ
Processor	Near Memory: 2.5 GHz out-of-order core Far Memory: 4 GHz out-of-order core
L1-I/D Cache	32KB, 8-way, 4 cycle access
L2 Cache	inclusive, private, 256KB, 8-way,11 cycle access
L3 Cache	inclusive, shared, 8MB, 16-way, 40 cycles
Interconnect	Split Bus, 6 cycles, arbitrate latency: 1 cycle
DRAM	4GB, 2 channels, tCL = tRCD = tRP = 13.75ns, tCk=1.25ns
3D Memory	4GB, 32 vaults [12], 128 TSV's per vault @2Gb/s [72]
Off-chip links	4 SerDes links, 16 lanes per link, direction [12]

Table 4.2: Processor and memory model.

Benchmark	LLC_MPKI	IPC	Benchmark	LLC_MPKI	IPC
povray	0.06	0.94	perlbench (perl)	1.42	1.20
gamess	0.1	1.33	gcc	1.49	0.65
namd	0.13	1.13	cactusADM (cactus)	3.58	0.71
hmmer	0.26	1.08	zeusmp	4.02	0.84
calculix	0.31	1.17	bwaves	10.32	0.69
gobmk	0.34	0.93	leslie3d (leslie)	17.53	0.38
h264ref	0.43	1.21	GemsFDTD (Gems)	20.25	0.27
gromacs	0.46	0.76	milc	20.58	0.45
sjeng	0.47	0.86	soplex	25.93	0.27
tonto	0.54	1.01	libquantum (libq)	33.06	0.32
bzip2	0.55	0.75	mcf	40.67	0.15

Table 4.3: LLC_MPKI and IPC for DRAM_hp.

representative execution samples. Table 4.3 reports the LLC misses per kilo instructions (MPKI) and IPC values for DRAM_hp (unsecure baseline without smart memory).

Processor Model: We modeled our processor designs (Table 4.2) using MARSSx86 [105], a full system cycle accurate simulator. Processor in InvisiMem_far is similar to Intel Quad Core i7-4790K processor [13]. InvisiMem_near places secure processor in the logic die of smart memory. Eckert et al. [51] investigated power dissipation possible in the logic layer of smart memory under various cooling solutions. They conclude that with an active heat sink, the power dissipated can be as high as 55W without affecting memory die tempera-

Design	Read-Req	Read-Resp	Write-Req	Write-Resp
Baseline	16	80	80	16
InvisiMem_far	112	120	112	120

Table 4.4: Request and response packet sizes (in bytes).

tures adversely. Hence, we model InvisiMem_near as Intel i7-3770T [11] at 2.5GHz and 45W.

Latency of Cryptographic Primitives: We synthesized a pipelined AES core from OpenCores [17] at 45nm and scaled it using ITRS projections to model its latency in our system. The Galois Field multiplication (authenticated encryption) is a combinational circuit that operates in single cycle [123, 152].

Power Model: We model processor power using McPAT [90] and AES energy to be 302 pJ [95] per 128-bit block. For baseline DRAM, we model access energy to be 65 pJ/bit [72]. A recent industry prototype [72] reports 10.48pJ/bit for HMC access of which 43% is attributed to SerDes circuits [72, 109], rest is for DRAM access and logic layer. We model 1.42W for DRAM static power.

Smart Memory Model: We use DRAMSIM2 [118] to model 4GB of DRAM memory for baseline (DRAM_hp). We modify DRAMSIM2 to model a 4GB 3D-stacked memory with 32 vaults and 128 TSVs (through silicon vias) per vault [12]. We assume the same DRAM device parameters (Table 4.2) for both traditional DRAM and 3D-stacked memory. However, we assume a DRAM clock in line with TSV signaling rate for smart memory [72].

Table 4.4 shows packet sizes communicated in baseline [12] and InvisiMem_far. Each packet has 8-bytes of header and tail, which carry useful meta-data like command, address

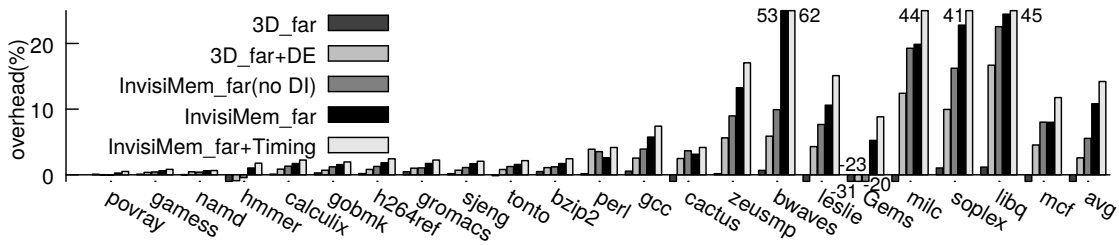


Figure 4.8: Performance overhead of far-memory processor unsecure and secure designs w.r.t DRAM_hp.

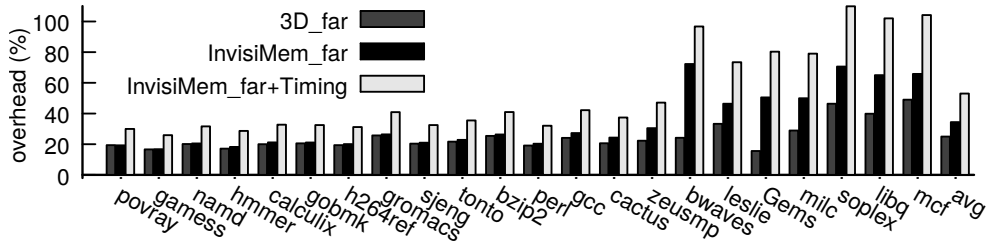


Figure 4.9: Energy overhead w.r.t DRAM_hp.

etc. Hence, read request or write response is 16 bytes in size. Write requests and read responses carry 64 bytes of data as well. In our design read/write requests also transmit authentication tags (16 bytes for packet header/tail, 16 bytes for data). Responses additionally carry data timestamp (8 bytes).

4.5.2 Unsecure Smart Memory Performance and Energy

Figure 4.8 shows the performance overhead with respect to unsecure baseline DRAM_-hp of various designs modeled with increasing security guarantees. We plot the benchmarks in the increasing order of their LLC miss rates. The 3D_far design represents an unsecure high power processor connected to smart memory. High bandwidth smart memory helps improve performance of memory intensive programs (`GemsFDTD` sees gain of 31.41%). On average, smart memory delivers performance improvement of 4.02%.

Figure 4.9 shows the energy overhead of 3D_far design (average 24.98%). While the

DRAM energy is lower for smart memory, the static power expended by the SerDes links is the chief cause of this overhead. High static power is caused as SerDes links transmit null packets when idle [12]. Prior work also observes that SerDes link power is a significant fraction of the total HMC power [21, 109, 59].

4.5.3 Far InvisiMem

This section discusses performance and energy overheads of our InvisiMem_far designs to guarantee security properties equivalent to ORAM, data integrity, freshness, and also avoid leaking timing of memory events.

3D_far+DE configuration in Figure 4.8 adds data encryption (DE) to 3D_far design. This model helps us tease out data encryption overheads (incurred in secure processors like Intel SGX) from the address encryption overheads. Adding data encryption incurs modest overhead of 2.58% on average.

To tease out the overhead of providing ORAM guarantees from the other security guarantees, we model InvisiMem_far (no DI) configuration which provides only ORAM guarantee. In this design we encrypt both address and data, but authenticate only address. This increases the overhead from 2.58% (3D_far+DE) to mere 5.55%.

The InvisiMem_far configuration depicts the design which has ORAM, data integrity and freshness guarantees, but no defense against the timing channel. InvisiMem_far design, incurs an average overhead of 10.81% (highest overhead for bwaves of 52.65%). This is a significant improvement over prior ORAM-based solutions [56], which also require additional hardware support for tracking and checking version numbers of memory blocks. The InvisiMem_far configuration which does not leak the timing of memory events is depicted

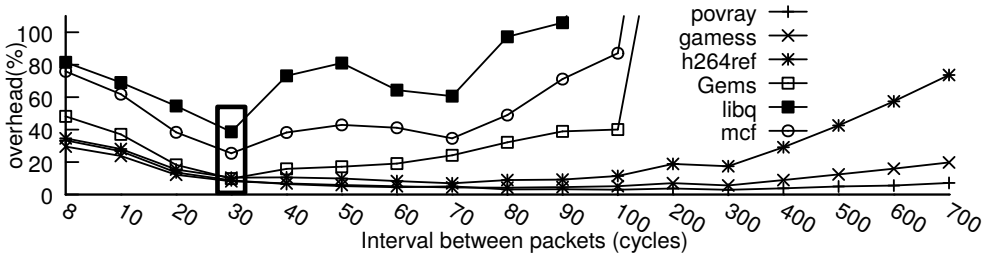


Figure 4.10: ED^2 overhead w.r.t InvisiMem_far without timing channel defense for various static memory access rates.

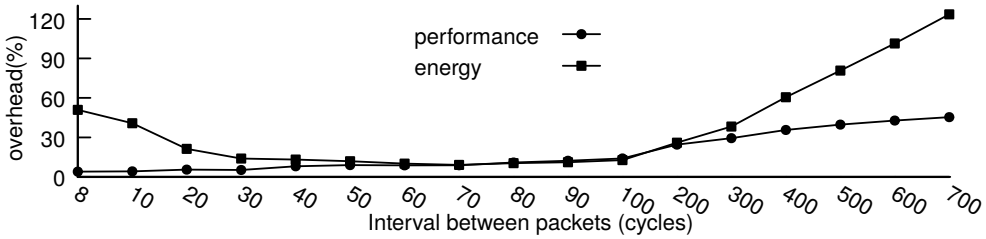


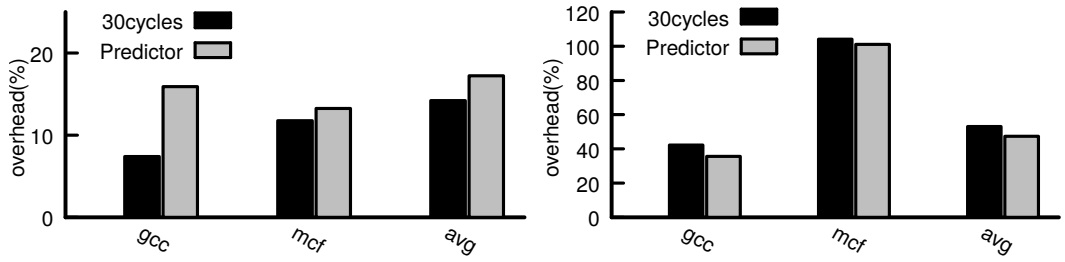
Figure 4.11: Overheads w.r.t InvisiMem_far without timing channel defense for various static memory access rates.

as InvisiMem_far+Timing (Section 4.5.4). This design increases the average overhead from 10.81% to 14.21%.

Figure 4.9 shows the energy overheads of InvisiMem_far. Without timing channel defense, InvisiMem_far increases the energy overhead of 3D_far design from 24.98% to 34.38%; with it the overhead is 53.03%. This is a sharp contrast to prior works which incur one to two orders of performance loss, bandwidth increase, and commensurate energy overhead.

4.5.4 Static Packet Rate for Timing Channel

As discussed in Section 4.3.4, we choose a static request and response rate to address timing channel leaks. We provide here the empirical evidence to support this choice. Figure 4.10 depicts energy delay squared (ED^2) overhead of various static packet rates with respect to InvisiMem_far without timing channel protection. To depict spectrum of behav-



(a) Performance (b) Total Energy

Figure 4.12: Comparison to dynamic scheme.

iors, we pick two least and most memory intensive benchmarks, and two benchmarks with highest and lowest IPC.

We see that the lowest (ED^2) overhead occurs roughly at 30-cycles for all these diverse programs. The reason is that energy consumed by cryptographic units to process dummy packets stops being a significant fraction of system power as packet interval increases beyond about 30-70 cycles. As SerDes links constantly send null packets even when they are idle, there is not much to be gained by increasing the packet interval beyond this sweet point. This combined effect is depicted in Figure 4.11 (averaged across six benchmarks under study) wherein energy overheads first start to drop before showing a negative trend.

We also implemented a dynamic predictor [57] (ED^2 overhead of 165.58%) with rates (30, 60, 120, 240). Figure 4.12 compares this predictor to our static scheme with 30 cycles (ED^2 overhead of 159.73%). We show low (`gcc`) and high (`mcf`) LLC_MPKI rate benchmarks, and also average for all the programs. As these results show there is not a significant potential for performance improvements or energy savings with a dynamic scheme. Considering it has a weaker security guarantee, we chose a static rate.

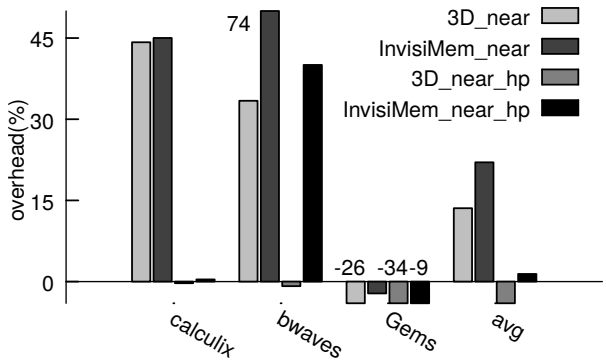


Figure 4.13: Performance overhead of near-memory designs w.r.t DRAM_hp.

4.5.5 Near InvisiMem

Figure 4.13 shows the performance overhead of two near-memory processor designs we model, with and without security guarantees (We depict outlier benchmarks only for space considerations). A low-power processor stacked with memory layers is depicted as 3D_near. Compared to high-performance far processor, it has an average overhead of 13.56%. Compute intensive benchmarks exhibit overheads (44.23% for `calculix`), whereas memory-intensive benchmarks see performance gains (`GemsFDTD`, 26.43%). InvisiMem_near which encrypts data and authenticates it on reads adds about 11% overhead to 3D_near design. To model the scenario where it may be feasible to connect a high performance core to memory through secure silicon interposer, we depict 3D_near_hp and InvisiMem_near_hp. For such a design, the average overhead of providing data encryption and integrity is a mere 1.41%.

4.5.6 Memory Space Overhead

Table 4.5 lists the space overheads of a recent ORAM-based proposal Freecursive-ORAM [56] and InvisiMem_far for various security guarantees. For both designs we report

Guarantee	Encryption	ORAM	Integrity
Freecursive ORAM [56]	256MB	5.8GB	2GB
InvisiMem_far	512MB	0B	1GB

Table 4.5: Memory space overheads.

space overheads for 4GB of real data with 64 bytes block size.

InvisiMem_far incurs more space overhead to store encryption timestamps as these are per cache block. In contrast, in Freecursive-ORAM, these timestamps are per bucket which comprises of multiple cache blocks. However, this reduction in space overhead has concomitant performance, energy and bandwidth cost as buckets have to be read and written in their entirety. InvisiMem_far incurs no space overhead for ORAM guarantees. Freecursive-ORAM, on the other hand, incurs close to 100% space overhead to store dummy data blocks and other metadata needed to implement the ORAM algorithm. Finally, for data integrity, Freecursive-ORAM has higher overheads as it needs tags for dummy cache blocks as well.

4.5.7 Fragmented Vs Non-Fragmented

Section 4.4.4 discussed two ways in which a memory block and its metadata (timestamp/tag) can be stored and retrieved from memory. Non-fragmented design has better memory utilization than fragmented design. However, it also breaks every memory request into requests for data and metadata. Owing to vault-level parallelism and our data mapping policy (Section 4.4.4) we see that the average overhead of InvisiMem_far only increases from 7.24% in fragmented design to 10.81% in non-fragmented design. Given its better memory utilization and negligible performance difference, we chose the memory layout of non-fragmented design for all our experiments.

4.6 Related Work

InvisiMem is the first work that uses smart memory based solution for memory bus side channel.

4.6.1 3D Stacking for Security

Only two prior proposals [140, 63] harness 3D-stacking to provide security guarantees. In [140] a control plane is integrated with a conventional processor in a 3D stack which provides security functionalities like monitoring activities of the processor to prevent cache-side channel attacks. In [63], the authors leverage smart memory logic to efficiently implement Bonsai Merkle Tree [117]. Our work obviates need for Merkle trees by using memory isolation provided by Intel SGX and employing authenticated encryption between processor and memory. Both prior works did not harness smart memories to solve address side channel or timing channel.

Concurrent to our work, ObfusMem [25] also uses smart memory to provide ORAM-equivalent guarantee. InvisiMem provides a stronger memory-trace obliviousness (MTO) guarantee [91] by hiding memory access and response times using constant rate messages.

4.6.2 Secure Hardware

Several secure hardware proposals [138, 9, 61, 136, 31] exist which chiefly aim to provide isolation (protect sensitive application from other applications and untrusted system software) and software attestation. The latest proposal: Intel SGX [97, 23], provides isolated execution, and reduces the trusted computing base to the application and few privi-

leged containers. It also provides encryption, data integrity and freshness guarantees. There are other proposals with similar or more security guarantees like SGX [36, 52, 40]. None of these proposals address memory bus side channel.

We discussed solutions [91, 160, 56] that provide defenses against memory bus side channel in Section 4.2. They incur order of magnitude more memory accesses and result in huge performance overheads. We show in this work that with smart memory, memory bus side channel can be solved with low overheads. Prior works have also considered sending memory requests from the processor at a static [55] or dynamic [57] rate. Both rely on ORAM algorithm to generate indistinguishable real and dummy requests. Our timing solution does not rely on ORAM-algorithm. By employing smart memory it can generate constant response rate to hide response time variations. It can avoid processing dummy requests in memory, saving energy. Also, unlike prior schemes [57], our solution is not limited to only one pending memory request at any time.

Prior works [143, 127] address information leakage when an untrusted program shares the memory system with a trusted application. Wang et al. [143] extended memory controller to allocate fixed time quantum for each thread when they can issue memory accesses. It does not hide when a thread issues requests or receives responses within its time quantum from an adversary who has physical access to memory bus. Shafiee et al. [127] proposed to issue a real or dummy request every Q cycles for each thread, and architected a deterministic memory that guaranteed a response before the interval ends. Deterministic memory forgoes optimizations such as row-buffer hits. Unlike InvisiMem, it does not hide type of memory access (read or write). Also, InvisiMem leverages packetized interface of smart memory to support constant rate responses without requiring any changes to DRAM de-

sign, and is more efficient.

Optimizing Memory Encryption: Prior works propose several optimizations to reduce memory encryption overheads which can be used in our design. In [130], the authors predict encryption counters for speculative OTP pre-computation. In [153, 152], encryption counters are cached which can also further reduce our overheads.

4.7 Conclusion

This chapter proposed InvisiMem, which harnesses smart memory with compute capability to simplify solutions for providing address and data confidentiality, data integrity, freshness, and also closes the memory bus timing channel. By including logic layer of smart memory in the trusted computing base, we demonstrate how each of the above guarantees can be obtained at order of magnitude lower overheads for performance, space, energy and memory bandwidth, when compared to prior solutions that relied on expensive constructs like Oblivious RAM and Merkle trees.

CHAPTER V

Sanctuary: Efficient Page Fault Channel Defense

Previous chapter discussed how an attacker can learn the address trace of an application by snooping the addresses off the bus and a low-overhead defense against this side-channel. A related vulnerability that commercial secure processors like Intel SGX suffer from is page fault side channel. A malicious OS uses the page fault mechanism to induce faults at each memory access to learn the address trace of the application. This chapter presents Sanctuary, a low-overhead page fault channel defense which provisions operating system with flexibility to manage memory as a resource, yet prevents the OS from using page faults to learn an application's address trace. Our solution relies on near-memory page movements to keep it's overheads low. Together with InvisiMem (Chapter IV), Sanctuary secures the address trace of an application from leaking and leads to more secure systems at low overheads.

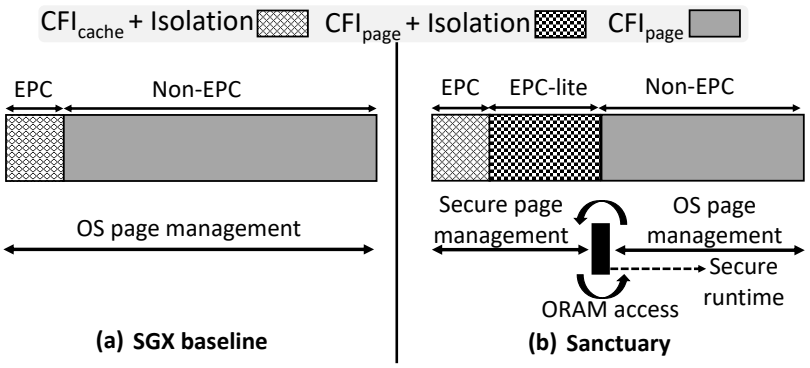


Figure 5.1: Memory organization under SGX (a) and under Sanctuary (b). CFI: Confidentiality, Freshness, Integrity

5.1 Introduction

Reliance on cloud computing for it's myriad benefits has led to increased demand for preserving the privacy of cloud computing client's data and computations. Intel Software Guard Extensions (SGX) [23, 97] is the latest commercially available secure processor offering which aims to answer the increased demand for privacy preserving remote computations. With SGX, a cloud user can designate parts of his application as private or sensitive (termed `enclave`) and the SGX-enabled processor will isolate code and data of this enclave from the rest of the system, including the application's public functions, system software, and hardware peripherals. A SGX enclave accesses its code/data by placing it in a partition of physical memory that SGX reserves at boot time termed *EPC* (Enclave Page Cache, Figure 5.1). Once a page in this partition is allocated to an enclave, SGX provisions checks to ensure this page is *isolated* from the rest system; *only the owing enclave can read/write to the page*.

The size of *EPC* is limited under SGX (128MB in current SGX processors) to keep metadata space and performance overheads under check which largely depend on *EPC* size.

Security guarantees for *EPC* are provided at cache-block granularity; every cache-block is encrypted (confidentiality), has associated MAC tag (integrity:prevent data corruption) and additional metadata to preserve freshness (read returns latest written value). Limited *EPC* size can cause sensitive pages to be spilled outside *EPC*. SGX provides similar security guarantees (Figure 5.1) to enclave's sensitive pages regardless of the memory partition they are stored in albeit at different granularities. When a page is being spilled from *EPC* to *non-EPC* memory, SGX encrypts the page for confidentiality, creates a single MAC tag for data integrity and uses a nonce for freshness. Unlike *EPC*, sensitive pages in *non-EPC* memory do not have isolation. In order for an enclave to access a spilled page, it has to be moved back to *EPC*. Such page movements are done by the OS using SGX instructions.

While the security guarantees provided by SGX for sensitive pages of an enclave are strong, SGX leaves page management entirely to the OS. The OS can allocate pages in *EPC* to enclaves, deallocate them at will and spill them to *non-EPC* memory. Furthermore, address translations (virtual to physical) are also under OS's purview. As such, OS handles TLB misses, accesses and updates page tables for enclaves.

As SGX leaves page management to the OS, a malicious OS can simply revoke page permissions to induce spurious page faults during enclave execution. Using this mechanism, the OS can learn the address trace of an enclave. This security vulnerability, termed page fault channel [151] can be used to recover sensitive inputs of an application [151, 163]. Specifically, prior work [151] showed how the sensitive input image to an enclave can be completely recovered. This can have catastrophic consequences say for a medical image processing cloud application.

Current solutions to fix page fault channel [40] propose to reserve all memory needed

by an enclave a priori and revoke OS's ability to perform address translations and memory deallocations during enclave's execution. This is undesirable from both enclave's and operating system's perspective. For the enclave, predetermining memory requirement is only possible either by severely limiting enclave's behavior (no dynamic memory allocations, no recursion etc.) or by reserving large amounts of memory a priori. The latter case can cause information leak if enclave memory requirement at runtime exceeds the reserved memory size leading to OS controlled paging activity. For the OS, disallowing memory deallocations robs the OS of it's flexibility in managing memory as a resource.

In this chapter, we propose Sanctuary, a page fault channel defense in which unlike prior solutions we allow OS to allocate memory on-demand and also deallocate memory during enclave's execution. Unlike baseline SGX, where OS is in complete control of page management for an enclave, under Sanctuary, while OS still retains complete control over allocations, *EPC* deallocations and page movements between *EPC* and *non-EPC* memory are performed by a secure runtime in collaboration with the OS. The runtime secures its transactions (hides addresses) with the OS via a novel construct termed *Oblivious Page Management* (OPAM) which is derived from Oblivious RAM (ORAM) [60] but is customized for the properties of page management for enclaves. The runtime is also responsible for securing address translations for enclave's sensitive pages. As OPAM transactions are costly, we reduce their number by creating a novel memory partition termed *EPC-lite* which has similar guarantees to *EPC* but does not incur the overheads of actually increasing *EPC* size.

Oblivious page management (OPAM): Under both baseline SGX and Sanctuary, OS retains the ability to deallocate *non-EPC* memory pages and swap them to a backing store.

A malicious OS can use this to revoke permissions on *non-EPC* memory pages and learn the address trace of an enclave. To prevent this, under Sanctuary, the secure runtime obfuscates the addresses accessed in *non-EPC* memory via ORAM construct.

ORAM is a cryptographic primitive which makes a memory access pattern computationally indistinguishable from a random access pattern of same length. To do so, it maintains a randomized mapping between each logical page and its physical location in memory (`position-map`) and also reshuffles pages in memory on each access. While several ORAM realizations exist, we focus on path-ORAM [135] in this work which organizes memory as a binary tree. We customize the traditional ORAM implementation for our context and term the resultant implementation oblivious page management (OPAM).

Reducing position-map overheads: A key structure which supports ORAM algorithm is `position-map` which stores mappings between a logical page and it's physical location in memory. Access to this structure is required for each ORAM access and is on the critical path. By encoding position map data inside page tables we show how we can do away with both the space and performance overheads of accessing this structure.

Dynamic ORAM: Prior solutions protect fixed memory size using ORAM construct. However, as we support on-demand memory allocation for an enclave, the amount of *non-EPC* memory needed by an enclave can grow/shrink over time necessitating an ORAM construct which supports this feature. While such dynamic ORAMs have been theoretically studied, our work realizes the first efficient implementation for them. We address several challenges that arise to support this feature like maintaining consistency of position map structure as the ORAM tree grows and also identify interesting opportunities like choosing which tree paths to grow to improve ORAM efficiency.

Dataless and efficient stash: Each access to ORAM tree is modeled as a read and write of all pages on a path in the tree. Due to the nature of the ORAM algorithm, on a path write, it is possible that not all read pages can be spilled back to the tree. Such failed spill pages are tracked in a `stash` structure. To keep the memory footprint low for stash, traditional algorithms periodically issue dummy ORAM accesses (access random paths in tree) which add to overheads. Sanctuary reduces stash footprint by making it dataless; on a spill failure, we simply track few bytes of metadata and pick a new victim page to spill. Further, each ORAM access also attempts to spill pages in stash and we come up with efficient mechanisms to do so for our context.

Thin nodes: Each ORAM access accesses multiple pages (accessed path) in order to hide the specific page which was accessed. While reducing the node width (pages per node) in an ORAM tree can reduce its overheads, such thin nodes are more susceptible to ORAM spill failures. We show how by smartly growing the ORAM tree and our dataless stash implementation help us to both reduce spill failures and be more resilient to them. This allows us to realize thin nodes and reap their lower overheads.

Near-memory page movements: The chief source of overhead for an ORAM access is that several pages are read and written on each ORAM access. The net effect of each such access is that either a page is spilled from *EPC* to *non-EPC* memory or fetched from *non-EPC* memory to *EPC* while rest of the pages are simply *moved* in *non-EPC* memory. Performing these page moves using traditional loads/stores will cause cache pollution besides incurring huge overheads. Instead, we perform these page movements in a hardware unit close to the memory controller. This accelerates these page moves, avoids cache pollution and lowers overhead of each ORAM access.

***EPC-lite* to reduce OPAM transactions:** While OPAM considerably reduces baseline ORAM algorithm overheads, each OPAM transaction is still costly. As these transactions are required only while accessing *non-EPC* memory, we could reduce their number by increasing *EPC*. Increasing *EPC* size, however, is challenging as SGX provides security guarantees for *EPC* pages at cache block granularity. As such, any increase in *EPC* size will incur additional performance and space overheads for maintaining and accessing the metadata needed for providing these guarantees. Instead, Sanctuary extends *EPC* without incurring metadata overheads by devising a memory partition which has all the security properties of *EPC* except at page-level. We term this novel memory partition as *EPC-lite*. Secure runtime can move pages between *EPC-lite* and *EPC* without needing OPAM transactions. We identify challenges in supporting *EPC-lite* region and propose simple solutions to address these challenges.

We model a suite of cloud computing applications: genome processing, vision applications, graph processing, and in-memory key value store which frequently process sensitive data including but not limited to medical images, genome sequences and social graphs. We demonstrate how page fault channel can be fixed with reasonable overheads for these applications.

5.2 Motivation and Background

5.2.1 Intel SGX

Intel SGX [23, 97] provisions instruction set extensions which rely on hardware support to provide isolated execution to an application. An application developer intending to create

an SGX application, first needs to identify data that is sensitive, data structures which hold this data and code that operates on these structures and place them in a separate trusted library termed as *enclave* [14]. SGX guarantees that as long as hardware is not tampered with, any memory claimed by the enclave is encrypted and its integrity is checked. SGX adds additional checks to prevent malicious system software (operating system, hypervisor) or other applications in the system from accessing the memory claimed by the enclave. While an adversary can corrupt the enclave memory using physical probing of memory, any such data corruption will be detected using integrity checks of SGX.

While a promising solution, Intel SGX is susceptible to several vulnerabilities. The vulnerability that we focus on in this work is page fault side channel [151]. As SGX leaves virtual memory management entirely under the purview of the OS, a malicious OS can manipulate page tables to induce spurious page faults and learn the page-level address trace of an application. Prior works [151] show how this channel can be used to recover sensitive inputs to an application.

5.2.2 Threat Model

We assume a secure processor with support for isolated execution like Intel SGX. Our attack model assumes a powerful adversary with full control over the operating system. We assume that the entire application is bundled as an enclave (along with necessary libraries) and it's interactions with external world are made secure i.e. the enclave is protected against attacks like Iago attacks [32]. We consider other side channels like cache [155], power [85], thermal [110], program execution time [158] outside the scope of this work.

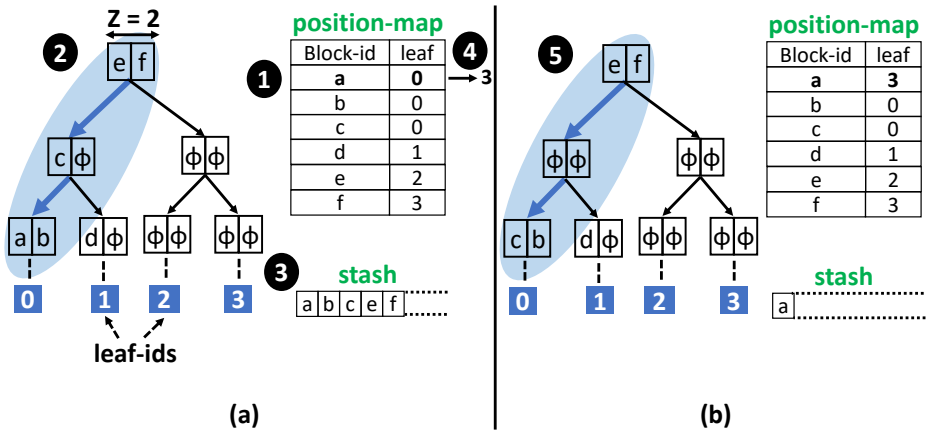


Figure 5.2: Accessing `block a` involves read and write of path to leaf to which the block is mapped. (a) Depicts stash after end of path read. (b) Depicts stash after end of path write. Block with dummy data is represented as ϕ.

5.2.3 Path Oblivious RAM

Oblivious RAM (ORAM) is a cryptographic construct which makes a memory trace computationally indistinguishable from a random access trace of same length. While there are several ORAM construct realizations, we employ path-ORAM [135] which is the most practical implementation of this construct. In this section we explain the workings of path-ORAM algorithm (henceforth referred to simply as ORAM).

ORAM organizes memory as a binary tree and each node in the tree has fixed number of slots (`z`) each capable of storing a single data block. The tree also has associated `utilization` factor which indicates percentage of real blocks that can be stored in the tree; remaining blocks hold dummy data. The algorithm maps each real block to a leaf in the tree and these mappings are maintained in the `position-map` structure. All the blocks (real and dummy) are stored in encrypted form in memory. The tree also stores metadata for every block which includes its block-id (null for dummy blocks), its leaf and encryption related counter.

Figure 5.2 depicts steps involved while reading `block a`. The algorithm first looks up the `position-map` to find the leaf the block is mapped to (❶). The ORAM invariant is that `block a` will either be in the `stash` structure the algorithm maintains or in some slot along the path to it's mapped leaf (`path-0`). Next, it accesses `path-0` and decrypts all blocks along the path (❷), storing only real blocks in the `stash` structure (❸). At this point, `block a` is read and remapped to a random leaf (❹). Next, as many blocks as possible are encrypted and written back to `path-0` and rest of the slots are filled with dummy blocks (❺). While writing back a path, data blocks are pushed as close to leaf nodes as possible (`block c` moves to leaf node). Notice that `block a` is left behind in the `stash`. This is so, as it is now mapped to leaf 3 and the only common node between `path-0` and `path-3` is the root node which is currently full. A write operation proceeds similarly except that the block will be read and updated.

ORAM's security relies on two actions. First, mapping of blocks to random leafs on each access causes new set of blocks to be read each time a block is accessed. Second, each access causes re-encryption of all blocks accessed which makes it hard for an adversary to differentiate between real and dummy blocks and deduce which block was actually accessed.

Note that ORAM is susceptible to failures. Most secure processor implementations provision `stash` as an on-chip structure and a stash overflow due accumulation of blocks causes ORAM failure. So as to reduce this probability, on every ORAM access, data blocks are pushed as close to leaf nodes as possible to free up higher level nodes which can store blocks mapped to larger set of leaves.

5.3 Sanctuary Design

This section gives details about three main components of Sanctuary's design: secure runtime, oblivious page management (OPAM) construct and *EPC-lite* memory.

5.3.1 Secure Runtime

The secure runtime is responsible for address translations for enclave's sensitive pages. All page faults incurred by an enclave are delivered to the secure runtime which performs the page table walk to read/update enclave's page tables. Further enclave's page tables are stored in enclave's sensitive pages so as to prevent malicious updates to them.

It is necessary to direct TLB misses to enclave's sensitive pages to enclave's page tables(stored securely and isolated) whereas misses to non-sensitive pages continue to use OS managed page tables. This necessitates enabling a dual page table walk mechanism. Such support has been extensively studied by prior work [40] and can be adopted in our system. In essence, this requires an additional page table base register which points to the physical address of enclave's page tables. Under SGX, programmer explicitly demarcates range of virtual addresses of an application as being sensitive and any address translation for this range can be directed to the enclave's page tables.

Under Sanctuary, the secure runtime also performs page management in collaboration with the OS. This includes making page management decisions on behalf of the enclave and securely performing page movements between *EPC* and *non-EPC* memory using our OPAM construct (Section 5.3.2). At enclave load, the secure runtime attempts to load the enclave's code and data section pages in *EPC*. If enough *EPC* is not present, these pages

are allocated in *non-EPC* memory. The secure runtime also reserves some pages for stack and heap sections. Profiling can be employed to figure out the number of pages to be set aside for these sections.

On a TLB miss, the secure runtime performs a page table walk. If the miss is due to a page present in *EPC*, the page table entry is simply loaded in TLB. If however the miss is due to a stack/heap page which was never allocated before, the secure runtime requests OS to allocate an *EPC* page. If a new *EPC* page is not available, the secure runtime picks a victim *EPC* page from the *EPC* pages allocated to the faulting enclave, spills it to *non-EPC* memory to make space in *EPC* and allocates it to the faulting address. Note that, during a spill to *non-EPC* memory, the secure runtime may need to request more *non-EPC* memory pages from the OS. If the miss is caused by a code/data section page or stack/heap page which was previously allocated but was spilled to *non-EPC* memory, the secure runtime will fetch the page in *EPC* which could also require an *EPC* page to spilled to make space for incoming *non-EPC* memory page.

5.3.2 Oblivious Page Management

In this section we discuss how we customize the traditional ORAM construct (Section 5.2.3) to be more suitable for page management context. We term our customized construct Oblivious Page Management (OPAM). Our secure runtime uses this construct to support page movements between *EPC* and *non-EPC* memory.

Sanctuary organizes the *non-EPC* memory pages belonging to an enclave as a binary tree as in traditional ORAM construct. The secure runtime can request allocations of *non-EPC* memory pages for an enclave. The OS can swap any *non-EPC* memory page to the

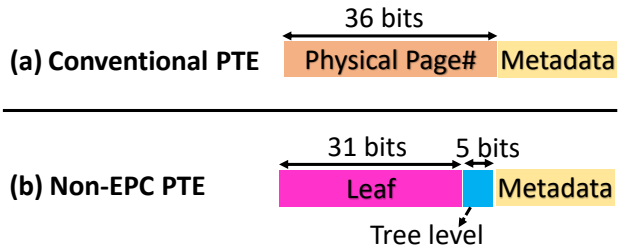


Figure 5.3: Sanctuary page table entries (PTE). A virtual address maps to *EPC* or *non-EPC* memory. For the former, PTE is as before (a) consisting of physical page number and PTE metadata. For the latter, we store leaf and tree level (b). This allows about 8TB of *non-EPC* memory per enclave.

backing store. However, on an access to the tree, the OS ensures all *non-EPC* memory pages on the accessed path are resident in memory and we provision mechanisms for the secure runtime to check this (Section 5.4.1.1). We discuss next our customizations to the traditional ORAM construct.

5.3.2.1 Page table as Position Map

Traditional ORAM algorithm requires a `position map` which tracks mapping between a page and leaf. One of the chief overheads of ORAM algorithm is the performance and space overheads associated with `position map`. We do away with these overheads by encoding `position map` data inside page table entries. On a TLB miss, page tables are already accessed and as such there is no performance overhead for accessing `position map` data. Figure 5.3 depicts this encoding where an enclave page is either mapped to *EPC* in which case the page table entry (PTE) stores conventional information (physical page number and PTE metadata). However, if the enclave page is mapped to *non-EPC* memory, the PTE stores the leaf to which the page is mapped. Assuming x86-based architecture, close to 36 bits are available of which we use 31 bits to store leaf information.

5.3.2.2 Dynamic ORAM

Traditional ORAM algorithm supports fixed memory size. In our context, predetermining total number of *non-EPC* memory pages needed by an enclave a priori is hard. Furthermore, reserving large number of *non-EPC* memory pages would imply a large OPAM tree. As the overhead of each OPAM access largely correlates to the tree size this would in turn would translate to large overheads. As a consequence, unlike in traditional ORAM, in our context it makes sense to grow the OPAM tree on-demand. While prior works [98] study such *dynamic* ORAMs theoretically, we identify several challenges in practically implementing this feature and present solutions to identified challenges. Note that, OPAM tree can also be shrunk as enclave frees memory. While desirable, we leave this to future work and only focus on growing the tree efficiently in this work.

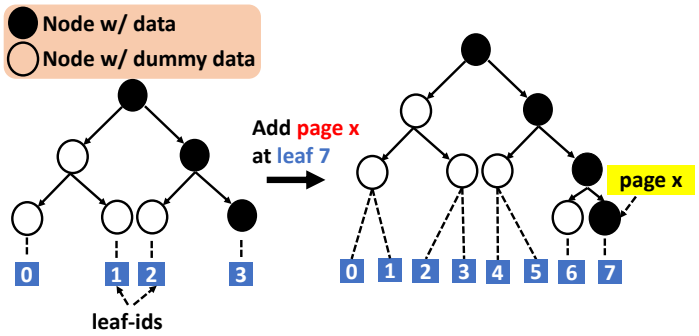


Figure 5.4: Smart growth. Adding nodes to a full tree (50% utilization). Naive growth adds nodes gradually from left to right which will cause the page addition to fail as path to leaf 7 is full. Smart growth prioritizes adding nodes to path which is accessed. As a consequence, the addition succeeds.

Growth Size: An interesting choice with dynamic ORAMs is when and by how much to grow the tree. Former is guided by the `utilization` factor (Section 5.2.3) which dictates the fraction of pages in the tree that can hold real data. Sanctuary adds nodes to

the tree at a spill to *non-EPC* memory which will cause the `utilization` factor to be exceeded. We add as many nodes so as to ensure that even after adding a new real page the `utilization` factor is not exceeded.

Smart Growth: While prior works [98] discuss gradual node addition to an ORAM tree, our work observes a unique opportunity that exists in deciding where to add these nodes. Figure 5.4 depicts a scenario wherein we are trying to add a page x to a full tree (utilization factor of 50%, has three real pages). As discussed above, at this point we will add two nodes to this tree so that post addition of page x the utilization factor is maintained. A naive growth strategy could add nodes left to right in a level and in this case cause the spill of page x to fail as path to leaf 7 is full.

Instead, in Sanctuary, we take a different approach which we term as Smart Growth. Under this optimization, when we attempt to grow the tree, we first try to add nodes to the path we are accessing. In this case, this causes nodes to be added to path to leaf 7 causing the spill of page x to succeed. If the path being accessed is already grown, we simply revert to adding nodes from left to right. We track a bit vector for last tree level to remember which paths have been grown so far and which need growing. Our evaluation shows that by prioritizing adding space to where it is needed the most, smart growth helps us reduce spill failures considerably.

Smart growth preserves the security of ORAM construct and the adversary does not learn anything new in smart growth as compared to naive growth. In both growth strategies the adversary only learns that the tree is being grown and which path in the tree is being accessed. By making node addition independent of current tree contents, smart growth

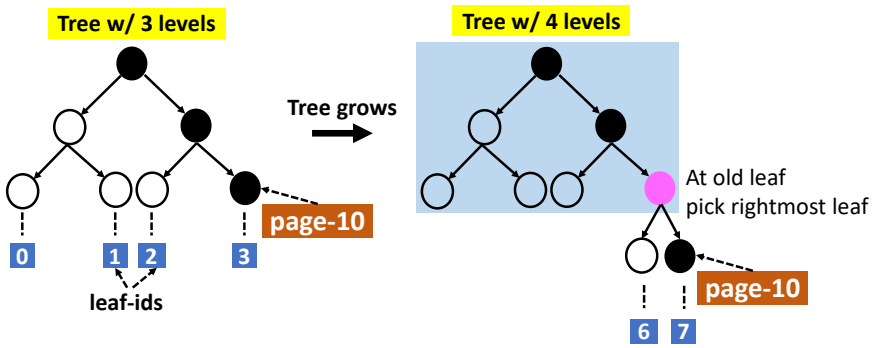


Figure 5.5: Finding page 10 in new tree which was mapped to leaf 3 in old tree. We employ deterministic remapping; even addresses get remapped to right paths and odd addresses to left paths. We also remember tree level with the mapping. To find the page, we find leaf in tree with #old levels and traverse the tree in relevant direction based on address.

does not leak any further information. Smart growth is independent of tree contents as regardless of occupancy of path being accessed, we still grow accessed path if we need to grow the tree and current path can be grown.

Avoiding Position Map Updates: One of the key challenge in realizing dynamic ORAM is position map consistency in which the leaf mapping stored in position map does not suffice in deducing which path in the tree is to be accessed. Position map consistency is affected in two scenarios: number of levels in the tree changes and when pages are shuffled at each access. Figure 5.5 depicts the former. When levels in a tree change, the #leafs in the tree change. The tree on the left has four leaves while on the right has eight leaves. The paths to leaf 3 in both the trees are vastly different. Also, page shuffling done on each access to the tree pushes data as close to leaf as possible. When the tree grows, two leafs are created where originally there was one leaf. We would like to spread old data over both leaves on page shuffling causing remapping of pages (update to leaf) needing position map updates. In the extreme case, we could iterate over all affected position map entries and

update them. This can have severe overheads as it will lead to random page table walks.

In Sanctuary, we employ two strategies which help us grow the tree and shuffle pages without needing to update the position map. First, we modify our position map entry to also store tree level as depicted in Figure 5.3. The leaf and tree level are stored the first time the entry it inserted in the page table. Also, during page shuffling, we employ deterministic remapping of pages. We remap pages with even addresses to even paths and odd addresses to odd paths. With these strategies, during accessing a page, we first use the levels stored in position map to identify leaf in the tree with stored #levels and then pick either the right-most leaf (even addresses) or left-most leaf (odd addresses) to identify the path to be accessed. Figure 5.5 shows an example.

5.3.2.3 Reducing Page Copies between EPC and non-EPC memory

In traditional ORAM implementations, on each tree access, all the pages on the accessed path are first decrypted and moved to `stash` (secure space) and then, as many pages in the `stash` as possible are written back. In our context, implementing this as-is would incur several page copies between *EPC* and *non-EPC* memory as the stash has to be in *EPC*. Such page copies are costly as SGX provides security guarantees at different granularity for *EPC* (cache-block level) and *non-EPC* memory (page level). As such, page copies have to transform metadata needed for these security guarantees (*non-EPC* memory to *EPC*: page level to cache-block level and vice versa). Note that, the net effect of tree access is that on each `spill` to *non-EPC* memory or `fetch` from *non-EPC* memory, only a single page actually needs to be copied between *EPC* and *non-EPC* memory while rest of the pages on the path only get shuffled in *non-EPC* memory.

So as to reduce these page copies, we harness our key observation that we only need to inspect OPAM metadata to deduce the page movement decisions and we need not move page data between *non-EPC* memory and *EPC* to get the net effect of an OPAM access. Recall that every page in OPAM tree has associated metadata (virtual address of page, leaf, level). We store this metadata separately from actual page data in a mirrored OPAM tree. We first access this data to deduce page copy decisions and subsequently copy one page between *EPC* and *non-EPC* memory (spill: *EPC* to *non-EPC* memory or fetch: *non-EPC* memory to *EPC*) and rest of the pages are simply moved within *non-EPC* memory (*non-EPC* memory to *non-EPC* memory). Our decoupling ensures that even with OPAM access, the page copies between *non-EPC* memory and *EPC* in Sanctuary are the same as in baseline SGX.

In order to support this feature, we need to enable additional primitives than currently available in SGX. SGX supports primitives for copying pages between *EPC* and *non-EPC* memory and vice versa. These primitives perform the necessary checks to ensure security guarantees (integrity and freshness). We need an additional primitive which copies data from an *non-EPC* memory page to another while performing these checks. In addition, in order to read OPAM metadata securely, we also need existing SGX primitives (*EPC* to *non-EPC* memory and *non-EPC* memory to *EPC*) to work at granularity smaller than a page. Prior works like Eleos [101] also propose having such primitives (sub-page access) so as to avoid moving page worth of data between *EPC* and *non-EPC* memory when locality is lacking.

5.3.2.4 Dataless and Efficient Stash

Recall from Section 5.2.3 that the ORAM algorithm uses a `stash` structure to track pages which could not be spilled on a path write. Prior ORAM implementations which model stash as a hardware structure have serious limits on how large this structure can be to limit on-chip storage. As such, when the number of blocks in stash increases to a certain threshold, these implementations need `dummy` ORAM accesses to empty the stash. A `dummy` ORAM access is simply an access to a random path in tree. The goal of such accesses is to find a path where some of the blocks in the stash can be spilled so as to reduce stash occupancy.

Sanctuary's realization of stash obviates the need for such dummy ORAM accesses and their concomitant overheads. In Sanctuary on a spill failure, we simply pick another page in *EPC* to spill and only remember small amount of metadata for the failed spill (virtual address of page, leaf, level). As such, our stash simply holds this metadata and is in effect dataless which reduces overall stash footprint. Furthermore, given our stash is dataless and in-memory we can also track larger number of spill failures.

On every ORAM access, attempt is made to spill data in stash to the tree (background spill processing). So as to push data as close to leaf nodes as possible, entries in stash are sorted based on the leaf currently being accessed. Tracking large number of entries in stash can cause high overheads for this sorting. In our design, we avoid this sorting. Instead we maintain a sorted list (by leaf-id) of available entries in stash which reduces stash processing to simple range checks. If an accessed path has a free slot, we compare the leaf range that can be spilled to this slot against leaf range available in stash. Only on

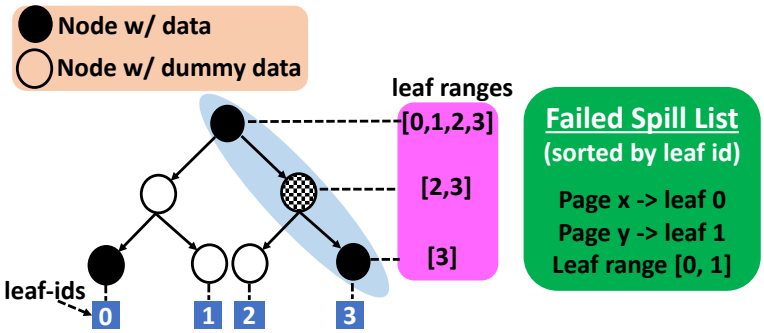


Figure 5.6: Fast background spill processing. We store past failed spills in a sorted order (by leaf id). Figure shows access of path to leaf 3 which has one empty node (checked) to which we can potentially spill a past failed spill. We also show the leaf ranges that can be spilled to each node. Simple range checks against these leaf ranges of available failed spills can help us process past failed spills quickly.

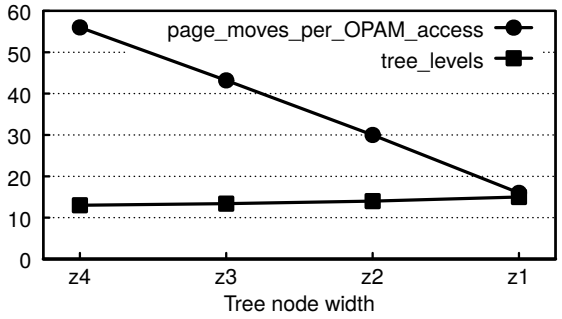


Figure 5.7: Thin nodes optimization: page movements reduce considerably with very small increase in tree height.

an overlap, do we check the stash for overlapped range only. Figure 5.6 shows an example.

5.3.2.5 Thin Nodes Optimization

Figure 5.7 depicts the effect of lowering ORAM tree node width (z). As the tree node width (z) decreases, the number of pages on the path also decrease, reducing page copies and overhead per access. However, as z decreases, the probability of spill failures also increases as the #options available to spill a page reduces (slots available on a path). Recall from Section 5.3.2.4 that such spill failures are tracked in stash and traditional implementations rely on dummy accesses to keep stash occupancy low. In fact, for tree node width 1, an

order of magnitude of more dummy accesses could be required making this configuration infeasible [111].

However, in Sanctuary, a confluence of optimizations makes thin nodes feasible. First, our stash is in-memory and is dataless making stash footprint very low. As such, we can track large number of spill failures. Furthermore, as discussed in Section 5.3.2.4 we also have an efficient mechanism to process stash. Finally, smart growth optimization (Section 5.3.2.2) keeps spill failures low. Together this helps us realize thin nodes and reduce OPAM access overheads.

5.3.2.6 Harnessing SGX Infrastructure

Recall from Section 5.2.3 that data in ORAM tree is stored in encrypted format and each access to the tree causes pages on the accessed path to be read and re-encrypted. Baseline SGX already has mechanisms to ensure confidentiality (encryption), integrity (MAC tag) and freshness (nonce) for every sensitive enclave page which is spilled to *non-EPC* memory. We use existing SGX mechanisms while reading and writing *non-EPC* memory pages as part of an OPAM access.

5.3.3 EPC-lite to reduce OPAM reliance

While our novel OPAM construct reduces the overhead of securing page movements between *EPC* and *non-EPC* memory pages considerably, each OPAM access causes several *non-EPC* memory pages to be copied and as such incurs high overheads. Recall that, we need to rely on OPAM only to access *non-EPC* memory pages. As such, we could reduce the number of OPAM accesses if we can increase *EPC* size.

The key impediment to increasing *EPC* size is the fact that SGX provides security guarantees for *EPC* pages at cache block granularity. As such, increasing *EPC* size causes a commensurate increase in space and performance overheads incurred for its metadata. Sanctuary overcomes this problem by extending *EPC* but providing security guarantees at page granularity. We term our *EPC* extension as *EPC-lite*. As *EPC-lite* has the same properties as *EPC*, page movements between *EPC-lite* and *EPC* do not use the OPAM construct.

Supporting *EPC-lite* requires extending baseline SGX isolation mechanism to also cover some *non-EPC* memory pages instead of only *EPC* pages. To support isolation for *EPC* pages, baseline SGX maintains a map structure (Enclave Page Cache Map, EPCM) which securely tracks some metadata for every *EPC* page. This metadata includes page permissions, the virtual address of page and also the enclave id who owns the page. SGX hardware checks this metadata to ensure only owing enclave issues reads/writes to an *EPC* page. These checks are performed each time a TLB miss resolves to an address in *EPC*. As *EPC* is a contiguous chunk of physical memory, checking if an address falls in *EPC* is a simple range check. To support isolation for pages in *EPC-lite*, we propose to track them similarly in EPCM. Also, just like baseline SGX, which marks *EPC* as no-DMA at memory controller, we also need similar support for *EPC-lite*.

Unlike *EPC* which is fixed at boot time, we propose that the *EPC-lite* be *dynamic*; the number of pages in *EPC-lite* can grow and shrink over time. This raises the question as to which *non-EPC* memory pages can be added to *EPC-lite*. If we provision support for any *non-EPC* memory page to be added to *EPC-lite*, every memory access needs to check EPCM which will add unnecessary overheads while accessing non-sensitive pages. In order

to avoid this, we constrain the *EPC-lite* region to be a contiguous memory chunk following the *EPC*. Doing so, preserves the single range check currently supported in baseline SGX.

While *EPC-lite* is interesting from a security standpoint, it does have associated limitations. Similar to *EPC*, deallocation of an *EPC-lite* page requires the request to be routed via the secure runtime. As such, larger the *EPC-lite* region, smaller the *non-EPC* memory which implies lower the control the OS has in moving pages to the backing store and more it's reliance on getting an enclave to deallocate an owned page. Also, constraining the *EPC-lite* to be a contiguous memory chunk as we do can lead to memory fragmentation as enclaves can give up *EPC-lite* pages as they finish execution. Some form of memory compaction in collaboration with the enclave's secure runtime will be necessary to reduce this fragmentation. We leave investigating this to future work.

5.4 Sanctuary Implementation

5.4.1 Sanctuary Metadata

Baseline SGX provisions metadata for every sensitive enclave page that is spilled to *non-EPC* memory which it uses to load the page back in *EPC* at a later time while ensuring security guarantees. We refer to this metadata as SGX metadata and this is different from the metadata needed per *non-EPC* memory page to support ORAM construct (termed OPAM metadata). We discuss in this section the changes to SGX metadata, OPAM metadata and how we separate the two and update/access them independently.

5.4.1.1 Changes to SGX Metadata

Baseline SGX creates metadata for an *EPC* page being spilled to *non-EPC* memory so as to be able to load the page back whilst ensuring confidentiality, integrity and freshness. This metadata includes host of information including (but not limited to) virtual address of the page, page permissions, enclave ID who owns the page and page type. Furthermore, SGX also encrypts the page, generates a MAC tag over both the page data and metadata and relies on a 8-byte `nonce` for freshness. Metadata so created except for nonce is stored in *non-EPC* memory along with the evicted page. Despite OPAM, if this metadata is left as is, a malicious OS can inspect it to learn which pages are being accessed. As a consequence, in Sanctuary, we also encrypt SGX metadata. Also, recall that some *non-EPC* memory pages are dummy. In order make real and dummy pages indistinguishable from each other we also we need SGX metadata (dummy) for these pages.

Sanctuary also needs to maintain the integrity of OPAM tree. While SGX ensures confidentiality, integrity and freshness for enclave's sensitive pages in *non-EPC* memory, we also need to ensure that the OS does not swap two nodes in the OPAM tree with impunity. To avoid such attacks, the MAC tag part of SGX metadata is also created over a unique `tree-id` which represents the position of the node in the tree. On an OPAM access, the secure runtime expects to read certain tree-ids and using this modified tag we can ensure the integrity of OPAM tree.

5.4.1.2 OPAM Metadata

We provision metadata per *non-EPC* memory page (real and dummy) which aids in the working of OPAM construct. We term this OPAM metadata and it includes the page's virtual address (VA), its leaf and level of OPAM tree. We need VA as the position map simply stores VA to leaf mapping and on accessing a path in tree we need to figure out which page matches the VA we are looking for. We need leaf and tree levels to support efficient page shuffling. Recall that on each access, pages are shuffled to push real data as close to leaf nodes as possible. Having leaf and level as part of metadata helps us perform page shuffling without having to access the position map.

As we inspect OPAM metadata independent of page data (Section 5.3.2.3), we store OPAM and SGX metadata separately. As such, we need to maintain security guarantees for OPAM metadata separately. Consequently, we encrypt this metadata and store a MAC tag for it. Similar to the tag that SGX stores (Section 5.4.1.1) this tag is also calculated over tree-id.

Recall that SGX relies on 8-byte nonces (stored securely) to ensure freshness for both *non-EPC* memory page and SGX metadata. We also need to provide similar guarantee for OPAM metadata. Straightforward addition of nonces for OPAM data can increase nonce overhead considerably. We optimize this overhead by observing that both OPAM and SGX metadata are updated on each page access albeit sequentially. As such, we repurpose existing SGX nonce storage for both the nonces: while OPAM metadata uses nonce, SGX metadata uses nonce+1.

5.4.2 OPAM Implementation

In this section we discuss some relevant implementation details for OPAM, additional hardware support we need for it and an optimization we employ.

We implement an OPAM tree with 50% utilization factor and in our evaluation we try different tree node widths (`z`). Each slot in our OPAM tree stores a single *non-EPC* memory page of size 4KB. Our OPAM tree supports three operations: `spill` which adds an *EPC* page to the tree, `fetch` which moves a *non-EPC* memory page to *EPC* and `increase` which increases the tree size by adding new *non-EPC* memory pages. Note that unlike traditional ORAM implementations, our OPAM tree is exclusive; a given logical page is either in the OPAM tree or in *EPC*. We do not attempt to hide the OPAM operation type (spill, fetch etc) from the adversary. However, we do hide the page access type (read/write).

5.4.2.1 Hardware Support for Additional Paging Primitives

As discussed in Section 5.3.2.3 we rely on new paging primitives which help us decouple page data and OPAM metadata access which in turn helps us reduce page copies done in our system. We describe the new primitives and the hardware support needed for them more concretely in this section.

Baseline SGX has support for moving a page worth of data between *EPC* and *non-EPC* memory. While moving a page from *EPC* to *non-EPC* memory, SGX encrypts the page, generates some metadata (Section 5.4.1.1) for the page, also generates MAC tag over page data and metadata and relies on a 8-byte nonce for freshness. While loading back the page from *non-EPC* memory to *EPC*, SGX checks the integrity and freshness of the page using

the tag and the nonce and only then writes it to *EPC*.

In our design, we first need an additional paging primitive which copies a *non-EPC* memory page to another after checking it's freshness and integrity. Further, we also need a primitive, which moves data between *EPC* and *non-EPC* memory at granularities smaller than a page while also performing integrity and freshness checks. We use this primitive to read and write OPAM metadata. While the latter primitive can be supported by adding granularity support to existing SGX primitives, for the former primitive, we need an entirely new primitive, which, while similar in spirit to existing primitives, instead reads a *non-EPC* memory page and writes to another.

Since these *non-EPC* memory pages are never accessed by the enclave post the OPAM access bringing them in caches can only pollute caches by kicking out useful data. Furthermore, using traditional loads/stores to perform these page moves will further exacerbate their overheads. Instead, we perform these page movements at memory controller periphery via a hardware unit close to the memory controller. This unit receives a list of source and destination page addresses, reads and decrypts page contents at source address, performs integrity and freshness checks just like in baseline SGX and then encrypts the page contents and writes them to destination page address along with generating new integrity tag and using updated freshness nonce.

5.4.2.2 Spill-ahead Optimization

As discussed in Section 5.3.1, unless the OS allocates a free *EPC* page to the enclave, a miss to a newly minted stack/heap page or previously spilled *EPC* page will require the enclave to pick a victim page amongst its *EPC* pages and spill it to *non-EPC* memory. This

implies, two OPAM accesses get added to the critical path (spill *EPC* page, fetch *non-EPC* memory page). Instead, we employ an optimization which helps us reduce the number of OPAM access to either zero or one. If we could have secure runtime running in a parallel thread, we could spill an *EPC* page ahead of time and keep a free *EPC* page at all times with an enclave. This can potentially take the spill of *EPC* page off the critical path. We term this as spill-ahead optimization and also evaluate such a configuration.

5.5 Applications and Security Context

We discuss in this section the cloud applications we study and also outline scenarios where these applications manipulate sensitive data.

- **Genome Processing:** We study PRIMEX [89], an open source genome sequence analysis algorithm which breaks up a given genome sequence into *k-mer*s (substrings of fixed length) and tracks their occurrences in the sequence in a table. This table then aids in quick searches over the sequence.

  Security Context: Genome data is highly sensitive as it can be used to identify a person, deduce if he is susceptible to any known diseases, ancestry information and much more. Given genome processing deals with large scale of data, cloud computing is often employed.

- **Graph Processing:** We study the following graph processing algorithms from GraphMat [137].

  PageRank: PageRank orders web pages based on some metric like popularity. Web pages are modeled as vertices and hyperlinks as edges and the algorithm scores each

vertex to determine its rank.

Breadth First Search (BFS): BFS takes a graph and an initial vertex and computes the distance (number of edges) to all reachable vertices from the initial vertex.

Single Source Shortest Path (SSSP): SSSP takes a weighted graph (each edge has corresponding weight) and an initial vertex and computes the minimum distance (using edge weights) of all vertices from the initial vertex.

Collaborative Filtering (SGD): This kernel is used by recommender systems [113] to deduce a given user's rating for a given item based on incomplete set of (user, item) ratings.

Security Context: Graphs are increasingly being employed in several domains including networks, natural language processing, social network analysis and bioinformatics. In response to this, several cloud-based graph analysis services have been made available to users including GraphLab, IBM System G, Dydra and more. Given their wide usage, graphs deal with wide range of sensitive data. Social network analysis [71] manipulates social graphs (containing sensitive information like political or personal views of people) and is used in disease transmission analysis and sociology. Graphs are also employed in bioinformatics to capture functional relationships between entities like genes and proteins.

- **Image Processing:** We study the following image processing applications from the San Diego Vision Benchmark Suite (SD-VBS) [141].

  Scale Invariant Feature Transform (SIFT): SIFT extracts features from images which are robust to scaling, rotation and noise. Features so extracted find variety

of uses in object recognition, panorama stitching, 3D scene modeling, tracking and many more.

Security context: SIFT is widely used in medical image analysis; an important step in diagnosis and subsequent treatment of diseases. SIFT aids in medical image registration [119], segmentation of medical images [121], stitching multiple medical images [35] and more.

Maximally Stable Extremal Regions (MSER): MSER is a method to detect blobs in images. MSER is used in 3D reconstruction from set of images [94], object and scene retrieval in videos e.g Video Google [134], street extraction from satellite images [120] and more.

Security Context: MSER is widely employed in visual surveillance [120] to aid in human detection and recognition, traffic analysis and vehicular tracking. MSER also finds utility in medical image segmentation [161].

- **Redis:** Redis [5] is an open source in-memory key-value data structure store. Redis supports complex data types likes sets, hashes, lists and sorted sets. In response to massive data explosion and bottlenecks of traditional databases, key-value stores like Redis are a solution of choice as is evident with their wide spread adoption (Amazon's SimpleDB, Google's AppEngine).

  Security Context: Key-value stores are often employed as caches for frequent computations like complex SQL queries over traditional databases. As such, they also manipulate a breadth of sensitive data from commercial (stock quotes, people location services) to medical (electronic health records) to military sectors.

5.6 Evaluation

In this section we demonstrate the efficacy of our proposed design and implementation. We first talk about how our OPAM design helps reduce overheads of secure page movements assuming currently supported *EPC* size. We then talk about how increasing *EPC* size as we do with *EPC-lite* optimization further helps reduce the cost of page fault channel defense.

5.6.1 Methodology

Benchmark	instructions	CPI	Benchmark	instructions	CPI
primex			pagerank		
yeast	6.2	1.42	amazon	32.6	0.67
worm	13.6	1.39	flickr	72.2	0.77
gorilla	24.8	1.41	wiki	178.1	0.75
redis			bfs		
4k1800s	4.8	1.67	amazon	12.6	0.86
4k3600s	7.8	1.68	flickr	32.7	0.93
4k7200s	12.1	1.68	wiki	71.9	0.92
sgd			sssp		
netflix_1	11.0	1.92	amazon	12.6	0.85
netflix_2	10.8	1.92	flickr	32.6	0.93
netflix_5	10.9	1.93	wiki	71.6	0.92
mser			sift		
hd	3.2	0.61	hd	20.1	0.41
sun	6.7	0.77	saturn	24.8	0.42
dog	18.2	1.39	sun_1	38.9	0.42
kme	26.8	1.26	sun_2	71.9	0.43

Table 5.1: Instructions (in billions) and CPI for unsecure baseline (native execution).

Application Inputs: For genome processing application `primex` we run the application with genome sequences having increasing sequence lengths from the Ensembl genome database [1]. For graph applications, we run real-world graph datasets (Amazon, Flickr and Wikipedia) from the University of Florida Sparse Matrix collection [46] and Netflix chal-

lenge for collaborative filtering [26]. For image processing applications (`mser, sift`) we model the largest dataset (full hd) from the San Diego Vision Benchmark Suite [141] and also run these applications with images from MIT-Adobe fivek dataset [4] to get larger memory footprints. We run `redis` using Memtier [6], a traffic pattern generator for key value stores for 4096 bytes object size for varying durations to get increasing memory footprints. Table 5.1 lists the instruction counts and CPI for unsecure baseline for the applications and the different input sizes.

Execution Model: We generate instruction level memory traces using PIN tool [92] which we use to infer instruction and data TLB misses. We model a 128 entry 4-way instruction TLB and a 64 entry 4-way data TLB. We use this TLB miss trace to infer *EPC* hits and misses. For *EPC* misses we infer the OPAM events that we incur. We model 96MB of *EPC* [1] based on current Intel SGX processors [62] and employ clock algorithm [30] for page replacement. We also study the OPAM events incurred and resultant overheads while modeling different *EPC-lite* sizes.

Performance Model: We use the OPAM events generated from the execution model to infer the performance overheads incurred by modeling both page movement and OPAM algorithm cost. We set the page movement (copy) cost assuming a standard memory system with 12.8 GB/s/channel and present results for a four channel memory system. We assume that OPAM algorithm cost is two times page movement cost. The algorithm cost involves reading metadata blocks and inferring page movement decisions. Note that this cost is much lower when tree levels are small and increases slowly as tree levels increase. Finally, we also assume that a parallel thread executes the OPAM events while inferring

[1] While actual *EPC* size is 128MB, only 96MB is usable and rest is used for metadata.

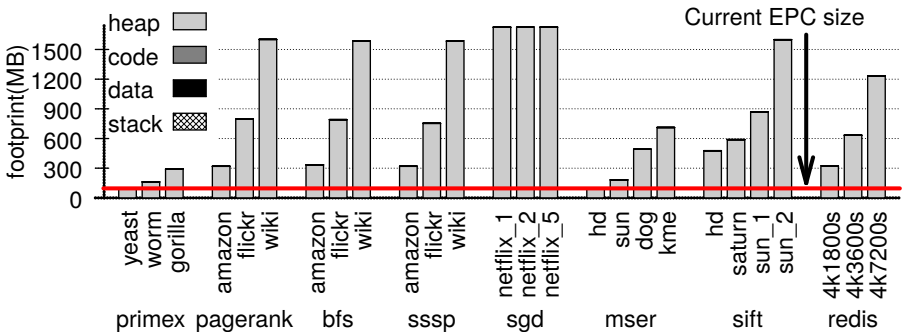


Figure 5.8: Memory footprint (accessed) of applications.

performance overheads. This helps us exploit the spill-ahead optimization (Section 5.4.2.2) which takes *EPC* spills off the critical path when possible.

5.6.2 Memory Footprint of Applications

We depict in Figure 5.8 the memory footprint of various applications we model for different input sizes. We track unique pages accessed by the application to deduce this footprint value and further divide it into code, data, stack and heap sections. For the applications we model, the memory footprint is dominated by heap pages. We pick inputs for the applications with increasing memory footprint size to evaluate how the overhead of fixing page fault channel changes as memory footprint increasingly exceeds *EPC* size. As an example, the footprint size for `pagerank` varies from 3X for `amazon` to 17X for `wiki` with respect to available *EPC* size. For `sgd` we do observe that changing the number of input movie files does lead to similar memory footprints.

Larger memory footprints are more likely to cause page movements across *EPC* and *non-EPC* memory boundary and as such could cause larger overheads. As our optimizations (OPAM and *EPC-lite* memory) reduce both the number of the page movements and cost of making them secure, the benefits of our optimizations will be more pronounced for

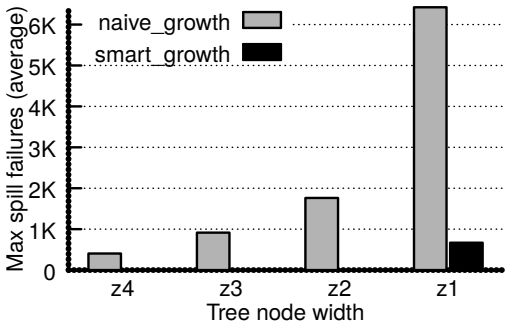


Figure 5.9: Comparison of maximum spill failures (average) for naive tree growth and smart growth. Smart tree growth considerably reduces spill failures by prioritizing accessed paths while adding space.

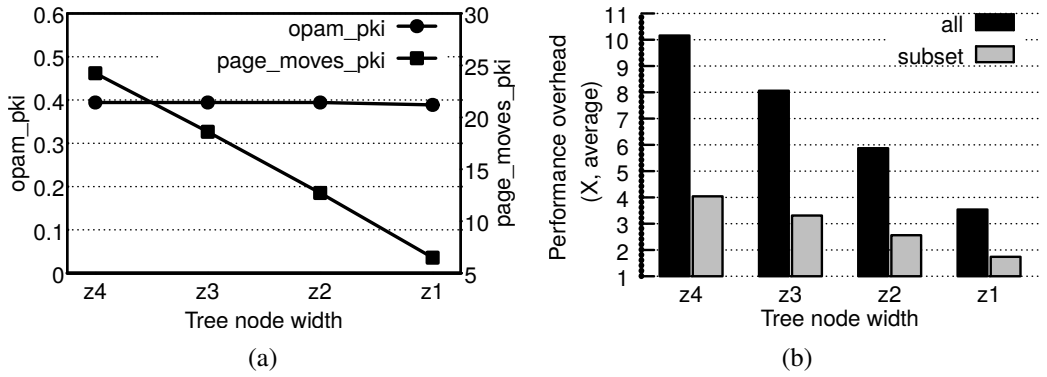


Figure 5.10: (a) Realized benefit of thin nodes optimization : page moves reduce considerably while OPAM events do not increase. (b) Performance overhead for existing *EPC* size (96MB) with thin nodes optimization.

larger memory footprints. The memory footprints we study in this work are largely limited by the simulation time needed to get traces for the entire application and the enormous storage needed for the resultant traces.

5.6.3 Evaluation of Smart Tree Growth

Figure 5.9 depicts the comparison of smart tree growth and its naive counterpart. We show the maximum spill failures (pending spill failures averaged across all applications) for different tree node widths (`z`). On a page eviction from *EPC* we randomly pick a path in the

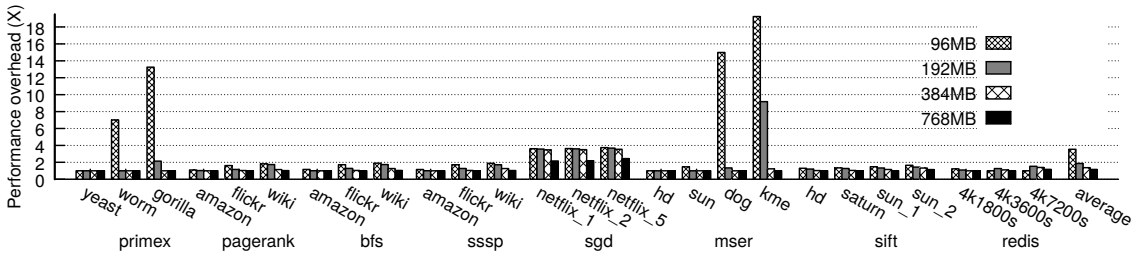


Figure 5.11: Performance overhead with a four channel memory system for increasing enclave-lite memory size (total isolated memory (usable) is depicted).

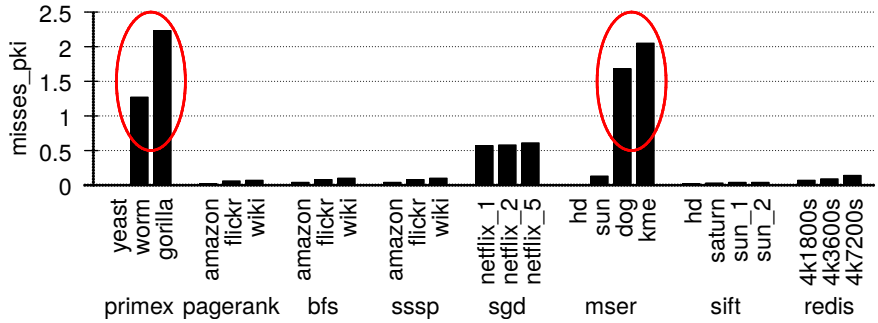


Figure 5.12: *EPC* misses per kilo instructions.

OPAM tree to spill this page. The lower the value of `z`, lower the options available along the chosen path and hence higher the chances of spill failures as is seen in Figure 5.9. While smart growth also depicts this behavior there is several orders of magnitude of reduction in the number of pending spill failures as compared to naive growth for higher values of `z` and close to an order of magnitude of reduction for `z1`. By prioritizing accessed paths, smart tree growth adds space to the OPAM tree where it is most needed and as a consequence far less failures need to be tracked and considered on each OPAM access.

5.6.4 Benefits of Thin Nodes

Thin nodes are interesting in that they help reduce performance cost of each OPAM event. As (`z`) decreases, while the performance cost of each OPAM event reduces (reduction in page moves needed), the spill failures also increase (Section 5.6.3) needing ability

to track these failures. In order to deal with increased spill failures prior works [111] incur an order of magnitude increase in OPAM events at lower values of (z) making such configurations infeasible. As discussed in Section 5.3, while smart growth helps us keep spill failures in check, dataless stash helps us keep the overhead of tracking spill failures low. Together they help us realize thin nodes optimization.

Figure 5.10a depicts both the OPAM events per kilo instructions and page moves per kilo instructions for different values of z averaged across all benchmarks. As we reduce z, the page moves needed per OPAM access reduces which will cause commensurate reduction in performance cost of each OPAM event. At the same time, the OPAM events needed do not increase as (z) decreases. In the following section we discuss how lower values of z help us reduce performance overheads.

5.6.5 Sanctuary Performance

5.6.5.1 Thin nodes optimization

In Figure 5.10b, we plot the performance overheads of Sanctuary as a consequence of OPAM events for different node widths as compared to a baseline which does not fix page fault channel. As discussed before, thin nodes reduce page moves needed for each OPAM event which is reflected in their lower performance overhead.

5.6.5.2 Enclave-lite optimization

We discussed in Section 5.3 how *EPC-lite* optimization helps us reduce the number of OPAM transactions needed. Figure 5.11 depicts performance overheads of Sanctuary as we increase *EPC-lite* memory size for a four channels memory system. As expected,

the performance overheads drop as memory size increases as the number of OPAM events drop and also as some of the workload's memory footprint fits within available memory. At 768MB memory, we see a performance overhead of mere 16% to fix page fault channel with memory footprint of fifteen of the available twenty-six workloads fits inside available memory.

5.6.5.3 Performance analysis

We also observe that, for a given application, the larger the delta between available *EPC* and *EPC-lite* memory and memory footprint of the application, more are the OPAM events incurred leading to increased overhead. However, the increase in overhead is not commensurate to memory footprint of the application. As an example, `sgd` has the highest memory footprint of all applications but not the highest performance overhead.

We observe that the OPAM events incurred by an application are more a property of its memory access behavior than its footprint. Figure 5.12 depicts the *EPC* miss rates observed per kilo instructions for applications under study. Some of the applications we model like `primex, mser` exhibit very high miss rates. We observe that these high miss rates translate to high performance overheads as depicted in Figure 5.11. Figure 5.10b depicts that the average overhead drops from 3.54X (`all`) to a mere 1.7X (`subset`) in absence of the workloads which exhibit high miss rates (`worm, gorilla, dog, kme`) assuming existing *EPC* size (96MB).

5.7 Related Work

5.7.1 Secure Hardware Proposals

In this section we talk about secure hardware proposals which contribute to trusted cloud computing. None of these solutions address the page fault channel and hence can benefit using our proposed designs. We talk about proposals which deal with fixing page fault channel in the next section.

Several proposals aim to build secure hardware to provide trusted cloud computing environments. The execute-only memory (XOM) architecture [138] creates compartments to isolate code and data of an application from other applications and system software. It assumes external memory is untrusted and so provides data encryption and integrity guarantees. However, it is susceptible to replay attacks. Trusted Platform Module(TPM) [9] provides a tamper resistant secure co-processor which is used to provide *software attestation*: a mechanism for a host to authenticate its software and hardware to a remote user. The attestation covers all software including host OS and thus, TPM trusts host OS to provide isolation to sensitive computation. Intels Trusted Execution Technology (TXT) [61] uses TPM co-processor for software attestation but reduces the software included in attestation to Virtual Machine (OS and application). The Aegis secure processor [136] trusts subset of host OS functionality (included in software attestation) to provide isolation guarantees to an application. The Aegis memory controller provides encryption and data integrity and is resistant to replay attacks. Bastion [31] relies on trusted hypervisor software (included in software attestation) to ensure isolation of an application from other applications and untrusted OS. The trusted hypervisor is invoked on every TLB miss to ensure this isolation.

Intel Software Guard Extensions (SGX) [97, 23] is the latest offering to provide isolated execution which reduces the amount of software included in software attestation to the application and few privileged containers. It also provides encryption, data integrity and freshness guarantees. Alternate proposals which provide SGX like security guarantees and/or add to it [36, 52, 40] have also been proposed. Given the commercial availability of SGX and the fact that it is already hardened against several attacks, we focus on fixing SGX's security vulnerabilities.

5.7.2 Prior Page Fault Channel Mitigations

Prior works which aim to tackle page fault channel exist. Like Sanctuary, Sanctum [40] secures address translations of enclave's sensitive pages but requires an enclave's memory requirements to be known a priori which is unrealistic. Also, once allocated, OS cannot reclaim memory from an enclave. T-SGX [131] breaks an enclave into several smaller transactions and relies on Intel Transactional Synchronization Extensions (Intel TSX) to get notified on a page fault. However, it assumes any page fault is a potential attack. This necessitates that enclave's entire memory footprint fit inside *EPC*. Given the small size of *EPC* on current SGX processors, this is also unrealistic.

In [133], the authors propose determinizing page fault access pattern by modifying each input dependent code/data page access to also pro-actively access several more pages. This incurs severe overheads (4000X) requiring manual program annotations and compiler analysis to lower them. Another approach also suggested in [133] requires enclave to declare a priori the set of pages it will access as a 'contract' and relies on processor to enforce no page faults to such pages. However, they only demonstrate their techniques for crypto-

graphic routines and not for general programs. Deja Vu [34] focuses on detecting privileged attacks. To do so, it creates a reference clock within an enclave and detects if enclave execution is longer than expected which signals a possible attack. Unlike Sanctuary, Deja Vu does not secure page movements between *EPC* and *non-EPC* memory.

5.7.3 Optimizing SGX Performance

Prior works [144, 101] which optimize SGX performance exist and can be adapted in our system. Both Hotcalls [144] and Eleos [101] identify frequent enclave exits on system calls as a performance bottleneck and eliminate them by offloading system call processing to a separate thread. Eleos [101] also isolates this thread from enclave threads in last level cache using Intel Cache Allocation Technology [38] to reduce LLC pollution caused by system calls. SCONE [24] enables secure containers with small TCB and lower overheads using SGX by relying on asynchronous system calls. Finally, Intel's SGX2 extensions [150] extend SGX so allow enclaves to dynamically add/remove pages. As part of this, they also provide support for Dynamic Regions which helps prevent frequent enclave exits by allowing an enclave to request a range of memory to be allocated with one call. Note that none of these works address the page fault channel.

5.8 Conclusion

In this chapter we presented Sanctuary, a page fault channel defense that preserves the flexibility operating system enjoys in managing memory as a resource. OS page management actions are routed through a secure runtime which uses a novel Oblivious Page Man-

agement (OPAM) construct to make transactions with OS secure. We use near-memory page movements to keep OPAM overhead low and also propose a novel memory partition *EPC-lite* which helps reduce the OPAM transactions needed and further helps reduce overheads of our solutions. Our results demonstrate that Sanctuary fixes page fault channel for a suite of cloud applications at reasonable overheads.

CHAPTER VI

Conclusion

We are deluged with tremendous amounts of data we observe and collect everyday. This data holds the key to understanding our world and making it better. Digitization of medical records can help spot patterns and aid in early diagnosis of diseases. Credit card companies can monitor transactions to identify fraudulent transactions. These are but few examples of the kinds of solutions one can expect by processing the massive amounts of data at our disposal.

This data explosion, however, exposes us to several new challenges. The research in this dissertation identifies two important challenges facing us and comes up with innovative architectures which tackle these challenges. First, we tackle the performance and energy inefficiency of conventional architectures via our proposal Compute Caches. Second, being cognizant of the increased demand for secure remote computation to process the massive amounts of data being produced, we come up with low-overhead hardware designs which defend against two important security vulnerabilities.

Today's compute centric systems incur high instruction processing overheads and expend significant energy over the deep memory hierarchies they employ. Consequently,

they are ill-equipped for data centric applications which process and analyze humongous amounts of data. To tackle this inefficiency, the research in this dissertation identifies a new near data processing technique which we term Compute Caches. Using emerging SRAM circuit technology of bitline computing, we make a case for caches that can compute. Until today, caches have served only as an intermediate low-latency storage unit. Our work directly challenges this conventional design paradigm, and proposes to impose a dual responsibility on caches: store *and* compute data. By doing so, we turn them into massively parallel vector units, and drastically reduce on chip data movement overhead.

We design simple, yet expressive, ISA extensions to expose Compute Cache functionalities. We identify and address several challenges in realizing an architecture where caches not only stage data but also compute. We re-designed a variety of data-centric applications, including text processing, database query processing (FastBit), cryptographic kernels, and OS checkpointing to leverage Compute Cache operations. Using Compute Caches as co-processors, we demonstrated significant speedup for this gamut of applications.

Another important challenge this dissertation focuses on is enabling secure remote computation. Cloud computing is increasingly being considered a viable, cost-effective way to process the large amounts of data we produce. However, users balk from adopting cloud computing because of security concerns. Increasing instances of data breaches and security attacks that we are witnessing only add to users' concerns. As such, the need for systems which provide better security guarantees is only going to increase. To this end, the research in this dissertation comes up with low-overhead hardware designs which defend against two important security vulnerabilities.

Memory bus side channel is an important vulnerability in secure hardware designs.

Merely by observing the addresses a program accesses, sensitive inputs to the program can be re-engineered. A practically feasible low-overhead hardware design that provides strong defenses against this vulnerability remains elusive. For example, current solutions to mitigate address side channel incurs about 100X memory bandwidth overhead, increase memory latency by over 20X and incur huge performance overheads (around 4X).

We instead propose InvisiMem, which uses compute capable smart memories to realize a practically feasible low-overhead solution to close memory bus side channel. We showed how emerging 3D stacked smart memory with packetized interface and logic layer close to memory can efficiently address memory bus side channel vulnerabilities without needing the ORAM construct. We also come up with a low-overhead solution to guarantee freshness using authenticated communication channel between processor and memory which obviates the need for expensive solutions like Merkel trees. We solve memory bus timing channel elegantly by sending heart-beat messages at a constant rate in both directions. Our work showed for the first time that hardware security is not just a processors responsibility, but memories have an important role to play. It closed the memory bus side channel using secure smart memory hardware that was several orders of magnitude more efficient than previously known solutions.

This dissertation also tackles a related vulnerability of page fault channel. A malicious OS can use the page fault mechanism to learn the address trace of an application. Commercial secure processor solutions like Intel SGX leave virtual memory management to the OS and are susceptible to this vulnerability. Prior works which try to fix this vulnerability assume that all application memory can be preallocated and do not let OS reclaim memory.

To overcome the limitations of prior solutions, this dissertation proposes Sanctuary, a

low-overhead solution which closes the page fault channel. We do so, without needing to preallocate all application memory apriori unlike prior works and still allowing the OS to reclaim allocated memory. We design a secure runtime which collaborates with the OS to perform page management functions on behalf of the application. We secure the runtime's interactions with the OS via a novel construct Oblivious page management (OPAM) which is derived from Oblivious RAM (ORAM) construct but adapted for page management context. We use near-memory page movements to keep OPAM overheads low. Finally, we also design a novel memory partition which helps reduce the OPAM transactions needed and further helps reduce overheads of our solutions. We study several interesting cloud applications which process sensitive data and demonstrate how we can tackle page fault channel with reasonable overheads.

To conclude, this dissertation identifies and addresses two important challenges that the data deluge we are facing exposes us to. We address the performance and energy inefficiency of conventional architectures in processing massive amounts of data via our proposal Compute Caches. We also address the increased demand for secure remote computation via low-overhead secure hardware designs: InvisiMem and Sanctuary. Together our proposals build a system which is better equipped to tackle massive amounts of data in a performance and energy efficient manner while also providing strong security guarantees.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Ensembl genome browser. "http://www.ensembl.org/index.html".

[2] Fastbit: An efficient compressed bitmap index technology. "https://sdm.lbl.gov/fastbit/,2015".

[3] IBM. IBM What is big data? Bringing big data to enterprise. http://www-01.ibm.com/software/data/bigdata/.

[4] Mit-adobe fivek dataset. "http://groups.csail.mit.edu/graphics/fivek_dataset/".

[5] Redis. "http://redis.io/".

[6] Redis labs. memtier benchmark. "https://github.com/RedisLabs/memtierbenchmark".

[7] The STAR experiment. http://www.star.bnl.gov/.

[8] Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual. version 1.2. Cray Inc., . 2003.

[9] Trusted computing group. tpm main specification. Retrieved April 1, 2016 from "http://www.trustedcomputinggroup.org/resources/tpm_main_specification,2003.", 2003.

[10] I2c bus monitor. Retrieved April 1, 2016 from "http://www.jupiteri.com/", 2006.

[11] Intel core i7-3770t. Retrieved April 1, 2016 from "http://ark.intel.com/products/65525/Intel-Core-i7-3770T-Processor-8M-Cache-up-to-3_70-GHz", 2012.

[12] Hybrid memory cube specification 2.0, 2014.

[13] Intel core i7-4790k. Retrieved April 1, 2016 from "http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz", 2014.

[14] Intel software guard extensions, enclave writer's guide, 2015. https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf.

[15] Xml parsing accelerator with intel streaming simd extensions 4 (intel sse4). Intel Developer Zone, 2015.

[16] Hmc. Retrieved April 1, 2017 from "https://www.micron.com/products/hybrid-memory-cube", 2016.

[17] Opencores. "http://opencores.org/", 2016.

[18] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinski, D. Blaauw, and T. Mudge. Scaling Towards Kilo-Core Processors with Asymmetric High Radix Topologies. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture (**HPCA-19**)*, 2013.

[19] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, 2000.

[20] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90.

[21] J. Ahn, S. Yoo, and K. Choi. Dynamic power management of off-chip links for hybrid memory cubes. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 139:1–139:6, 2014.

[22] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[23] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13.

[24] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza,

P. Pietzuch, and C. Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16.

[25] A. Awad, Y. Wang, D. Shands, and Y. Solihin. Obfusmem: A low-overhead access obfuscation for trusted memories. In *2017 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[26] J. Bennett and S. Lanning. The netflix prize. in kdd cup and workshop at acm sigkdd, 2007.

[27] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, 1996.

[28] M. Calhoun, S. Rixner, and A. Cox. Optimizing kernel block memory operations. In *Workshop on Memory Performance Issues*, 2006.

[29] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011.

[30] R. W. Carr and J. L. Hennessy. Wsclock—a simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81.

[31] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*.

[32] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[33] J. Chen and G. Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 216–228, 2014.

[34] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, 2017.

[35] W. Cheung and G. Hamarneh. n -sift: n -dimensional scale invariant feature transform. *IEEE Transactions on Image Processing*, 2009.

[36] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. Secureme: A hardware-software approach to full system security. In *Proceedings of the International Conference on Supercomputing*, ICS '11.

[37] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of*

the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, 2003.

[38] I. corp. Improving real-time performance by utilizing cache allocation technology. intel white paper., 2015.

[39] V. Costan and S. Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[40] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. Cryptology ePrint Archive, Report 2015/564, 2015.

[41] B. Dally. Power, programmability, and granularity: The challenges of exascale computing. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011.

[42] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA-19)*, 2013.

[43] R. Das, S. Eachempati, A. K. Mishra, D. Park, V. Narayanan, R. Iyer, and C. R. Das. Design and Evaluation of Hierarchical On-Chip Network Topologies for next generation CMPs. In *HPCA-15*, 2009.

[44] R. Das, S. Narayanasamy, S. Satpathy, and R. G. Dreslinski. Catnap: Energy Proportional Multiple Network-on-Chip Architecture. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA-40)*, 2013.

[45] S. Das, T. M. Aamodt, and W. J. Dally. Slip: Reducing wire energy in the memory hierarchy. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[46] T. Davis. The university of florida sparse matrix collection. "`http://www.cise.ufl.edu/research/sparse/matrices`".

[47] Y. Deng and W. P. Maly. Interconnect characteristics of 2.5-d system integration scheme. In *Proceedings of the 2001 International Symposium on Physical Design*, ISPD '01.

[48] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, pages 644–654, 1976.

[49] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02.

[50] F. Duarte and S. Wong. Cache-based memory copy hardware accelerator for multicore systems. *Computers, IEEE Transactions on*, 59(11), 2010.

[51] Y. Eckert, N. Jayasena, and G. Loh. Thermal feasibility of die-stacked processing in memory. In *Workshop on Near-Data Processing*, 2014.

[52] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceed-*

ings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47.

[53] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[54] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, 2006.

[55] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12.

[56] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. ASPLOS '15, 2015.

[57] C. W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, 2014.

[58] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 1995.

[59] M. Gokhale, S. Lloyd, and C. Macaraeg. Hybrid memory cube performance characterization on data-centric workloads. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pages 7:1–7:8, 2015.

[60] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*.

[61] D. Grawrock. Dynamics of a trusted platform: A building block approach. Retrieved April 1, 2016 from Intel Press, 2009, 2009.

[62] S. Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. `http://eprint.iacr.org/2016/204`.

[63] A. Gundu, A. S. Ardestani, M. Shevgoor, and R. Balasubramonian. A case for near data security. In *Workshop on Near-Data Processing*, 2014.

[64] Q. Guo, X. Guo, Y. Bai, and E. İpek. A resistive tcam accelerator for data-intensive computing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2011.

[65] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys.

[66] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[67] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*.

[68] Z. Huang and S. Li. A low power rsa design for smartcard. In *Proceedings of the 2011 Third International Workshop on Education Technology and Computer Science - Volume 01*, ETCS '11, pages 31–34, 2011.

[69] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[70] J. Jalminger and P. Stenstrom. A novel approach to cache block reuse predictions. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 2003.

[71] M. Jamali and H. Abolhassani. Different aspects of social network analysis. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '06.

[72] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*.

[73] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 2016.

[74] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu. Run-time cache bypassing. *Computers, IEEE Transactions on*, 1999.

[75] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15.

[76] M. Kang, E. P. Kim, M. s. Keel, and N. R. Shanbhag. Energy-efficient and high throughput sparse distributed memory architecture. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015.

[77] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. Flexram: toward an advanced intelligent memory system. In *Computer Design, 1999. (ICCD '99) International Conference on*, 1999.

[78] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, 2007.

[79] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 1992.

[80] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, New York, NY, USA.

[81] G. Kim, J. Kim, J. H. Ahn, and J. Kim. Memory-centric system interconnect design with hybrid memory cubes. PACT '13.

[82] J. Kim, J. Balfour, and W. Dally. Flattened butterfly topology for on-chip networks. *MICRO-40*, 2007.

[83] J. Kim and Y. Kim. Hbm: Memory solution for bandwidth-hungry processors. HotChips '14, 2014.

[84] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14.

[85] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 2011.

[86] P. Kogge. Execube-a new architecture for scaleable mpps. In *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, 1994.

[87] P. A. La Fratta and P. M. Kogge. Design enhancements for in-cache computations. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2009.

[88] O. L. Lempel. 2nd generation intel core processor family:intel core i7, i5 and i3. HotChips '11, 2011.

[89] M. Lexa and G. Valle. Primex: rapid identification of oligonucleotide matches in whole genomes. *Bioinformatics*, 19, 2003.

[90] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009.

[91] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. ASPLOS '15, 2015.

[92] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[93] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. CCS '13, 2013.

[94] D. Martinec and T. Pajdla. Consistent multi-view reconstruction from epipolar geometries with outliers. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*.

[95] S. Mathew, F. Sheikh, A. Agarwal, M. Kounavis, S. Hsu, H. Kaul, M. Anders, and R. Krishnamurthy. 53 gbps native $rmGF(2^4)^2$ composite-field aes-encrypt/decrypt accelerator for content-protection in 45 nm high-performance microprocessors. pages 767–776, 2011.

[96] D. A. McGrew and J. Viega. The galois/counter mode of operation (gcm). "http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf".

[97] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13.

[98] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan. Resizable tree-based oblivious ram. Cryptology ePrint Archive, Report 2014/732, 2014. http://eprint.iacr.org/2014/732.

[99] N. Muralimanohar. *Wire aware cache architecture*. PhD thesis, The University of Utah, http://www.cs.utah.edu/ rajeev/pubs/naveen-thesis.pdf, 2009.

[100] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1570–1581, 2015.

[101] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[102] M. Oskin, F. Chong, and T. Sherwood. Active pages: a computation model for

intelligent memory. In *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, 1998.

[103] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (cam) circuits and architectures: a tutorial and survey. *Solid-State Circuits, IEEE Journal of*, 2006.

[104] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim. Location-aware cache management for many-core processors with deep cache hierarchy. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[105] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[106] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *Micro, IEEE*, 1997.

[107] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *Micro, IEEE*, 1997.

[108] P. Pouliquen, A. Andreou, K. Strohbehn, and R. Jenkins. An associative memory integrated system for character recognition. In *Circuits and Systems, 1993., Proceedings of the 36th Midwest Symposium on*, 1993.

[109] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[110] J.-J. Quisquater and D. Samyde. Side channel cryptanalysis. In *Workshop on the Security of Communications on the Internet (SECI)*, 2002.

[111] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. ISCA '13, 2013.

[112] M. Rezaei and K. M. Kavi. Intelligent memory manager: Reducing cache pollution due to memory management functions. *J. Syst. Archit.*

[113] F. Ricci, L. Rokach, and B. Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011.

[114] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, 1998.

[115] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing*, 1998.

[116] R. L. Rivest and A. T. Sherman. *Advances in Cryptology: Proceedings of Crypto 82*, chapter Randomized Encryption Techniques. 1983.

[117] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. MICRO 40, 2007.

[118] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*

[119] Y. Sa. Medical image registration algorithm based on compressive sensing and scale-invariant feature transform. In *2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, 2015.

[120] E. Salahat, H. Saleh, A. S. Sluzek, B. Mohammad, M. Al-Qutayri, and M. Ismail. Novel mser-guided street extraction from satellite images. In *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*.

[121] L. A. Salazar-Licea, C. Mendoza, M. A. Aceves, J. C. Pedraza, and A. Pastrana-Palma. Automatic segmentation of mammograms using a scale-invariant feature transform and k-means clustering algorithm. In *2014 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*.

[122] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley. Cooperative cache scrubbing. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014.

[123] A. Satoh. High-speed parallel hardware architecture for galois counter mode. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, 2007.

[124] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud. Technical Report MSR-TR-2014-39, February 2014.

[125] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry. Fast bulk bitwise and and or in dram. *Computer Architecture Letters*, 2015.

[126] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46.

[127] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.

[128] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, 1999.

[129] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, 2002.

[130] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter

mode security architecture via prediction and precomputation. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 14–24, 2005.

[131] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26-March 1, 2017*, 2017.

[132] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *CoRR*, 2015.

[133] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, 2016.

[134] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*.

[135] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13.

[136] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03.

[137] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*

[138] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, 2000.

[139] A. Udipi, N. Muralimanohar, and R. Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010.

[140] J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner, T. Huffmire, C. Irvine, and T. Levin. Hardware assistance for trustworthy systems through 3-d integration. ACSAC '10, 2010.

[141] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. Sd-vbs: The san diego vision benchmark suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*.

[142] H. Wang, L.-S. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. In *MICRO*, 2003.

[143] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, 2014.

[144] O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[145] L. Whetsel. An ieee 1149.1 based logic/signature analyzer in a chip. In *Test Conference, 1991, Proceedings., International*, Oct 1991.

[146] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli. Calculating architectural vulnerability factors for spatial multi-bit transient faults. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[147] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, 1995.

[148] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2011.

[149] J.-J. Wu, Y.-H. Chen, M.-F. Chang, P.-W. Chou, C.-Y. Chen, H.-J. Liao, M.-B. Chen, Y.-H. Chu, W.-C. Wu, and H. Yamauchi. A large sigma vth vdd tolerant zigzag 8t sram with area-efficient decoupled differential sensing and fast write-back scheme. *Solid-State Circuits, IEEE Journal of*, 2011.

[150] B. C. Xing, M. Shanahan, and R. Leslie-Hurd. Intel® software guard extensions

(intel® sgx) software support for dynamic memory allocation inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, 2016.

[151] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *In Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[152] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*.

[153] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 351–360, 2003.

[154] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. OOPSLA '11.

[155] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, 2014.

[156] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, 2013.

[157] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, 2009.

[158] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[159] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, 2014.

[160] X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di. Fork path: Improving efficiency of oram by removing redundant memory accesses. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48.

[161] H. Zhu, J. Sheng, F. Zhang, J. Zhou, and J. Wang. Improved maximally stable extremal regions based method for the segmentation of ultrasonic liver images. *Multimedia Tools Appl.*

[162] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, 2013.

[163] X. Zhuang, T. Zhang, and S. Pande. Hide: An infrastructure for efficiently protecting

information leakage on the address bus. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, 2004.